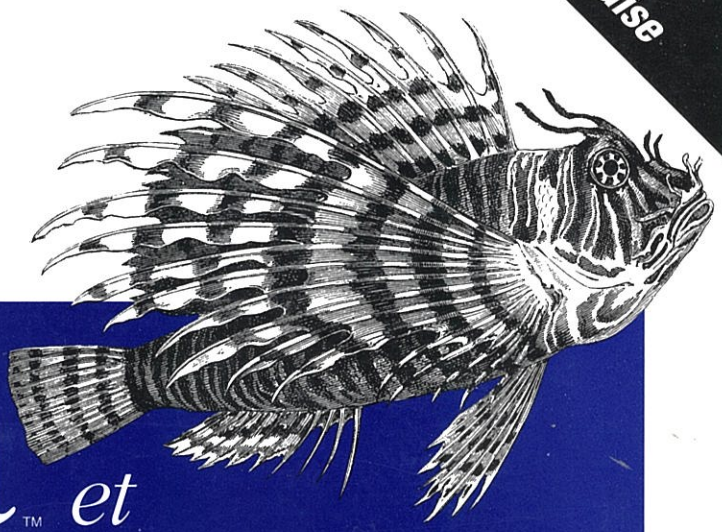
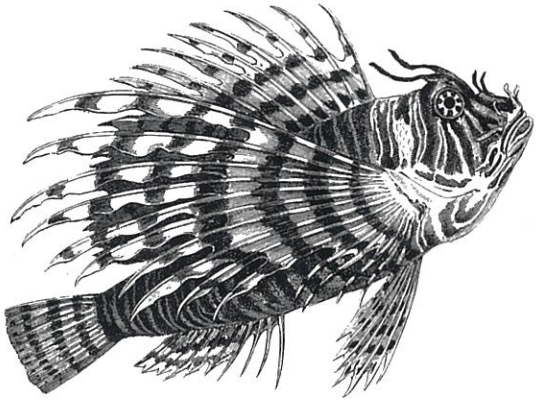
Développement de services Web en Java

Édition française



Java™ et SOAP



O'REILLY®

Robert Englander
Traduction de Dalil Djidel

Java et SOAP

Robert ENGLANDER

Java et SOAP

Traduction de
Dalil DJIDEL

Éditions O'REILLY
18 rue Séguier
75006 Paris
FRANCE
france@oreilly.com
http://www.oreilly.fr

O'REILLY®

Cambridge • Cologne • Farnham • Paris • Pékin • Sebastopol • Taipei • Tokyo

L'édition originale de ce livre (ISBN 0-596-00175-4) a été publiée aux États-Unis par O'Reilly & Associates sous le titre *Java™ and SOAP.*

© O'REILLY & ASSOCIATES, 2002

Couverture conçue par Emma COLBY, Hanna DYER *et* Melanie WANG

Édition : Dominique BURAUD

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

© ÉDITIONS O'REILLY, Paris, 2003
ISBN 2-84177-230-6

Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, de ses ayants droit, ou ayants cause, est illicite (loi du 11 mars 1957, alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 autorise uniquement, aux termes des alinéas 2 et 3 de l'article 41, les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part et, d'autre part, les analyses et les courtes citations dans un but d'exemple et d'illustration.

Une fois encore, pour ma fille Jessica.

Table des matières

Préface ..... *xi*

1. ***Introduction*** *1*
RPC et systèmes distribués orientés messages *2*
Données autodescriptives *2*
XML *4*
Spécifications de type API contre spécifications de type transmission *4*
Vue d'ensemble de SOAP *5*
Implémentations de SOAP *6*
Notre approche *7*
Comment commencer *8*

2. ***Messages SOAP*** *9*
La liaison HTTP *9*
Les requêtes HTTP *10*
La réponse HTTP *11*
L'enveloppe SOAP *13*
L'élément Envelope *16*
L'élément Header *16*
L'attribut actor *17*
L'attribut mustUnderstand *18*
L'attribut encodingStyle *19*
Gestion des versions de l'enveloppe *20*
L'élément Body *20*
Les fautes SOAP *21*

3. ***Encodage des données SOAP*** *25*
Schémas et espaces de noms *25*
Règles de sérialisation *28*
Indication du type *31*
Valeurs par défaut *42*
L'attribut SOAP Root *43*

4. *Services de type RPC* 45
Éléments RPC SOAP 45
Un service élémentaire 51
Déploiement du service 52
Écriture de clients pour les services 62
Déploiement avec une portée de niveau requête 70
Déploiement avec une portée de niveau session 72
Passage de paramètres 74

5. *Utilisation des types de données complexes* 85
Passage de tableaux en paramètres 85
Renvoyer des tableaux 94
Passage de types personnalisés en paramètres 98
Renvoyer des types personnalisés 109

6. *Sérialisation personnalisée* 117
Encodage de types personnalisés 117

7. *Fautes et exceptions* 143
Envoyer des exceptions côté serveur sous Apache SOAP 143
Créer un module d'écoute de fautes sous Apache SOAP 147
Envoyer et intercepter des exceptions sous GLUE 152

8. *Autres techniques* 157
Système de messages SOAP 158
Encodage littéral 168

9. *Interopérabilité et WSDL* 183
WSDL 184
Appeler un service GLUE depuis un client ApacheSOAP 193
Service d'intermédiation utilisant Apache SOAP 199
Appeler un service Apache SOAP depuis un client GLUE 204
Accéder à des services .NET 210
Écrire un client Apache Axis 216

10. *En-têtes SOAP* 219
Fournisseurs et routeurs Apache SOAP 220
Remplacer des classes fournisseur et routeur 220
Service Apache SOAP prenant en charge les en-têtes SOAP 225

11. *JAX-RPC et JAXM* 233
JAX-RPC 234
Travailler sans Ant 235
Créer un service JAX-RPC 236
Créer un client JAX-RPC 242
Générer des stubs par WSDL 243
DII (Dynamic Invocation Interface) 245
JAXM en résumé 246
Et ensuite ? 246

Index 247

Préface

SOAP, *Simple Object Access Protocol*, est le dernier protocole d'une longue lignée de technologies destinées à l'informatique distribuée. Il diffère des autres dans le fait qu'il est basé sur XML et qu'il n'a également pas pour l'instant l'ambition de révolutionner le monde de l'informatique. Au lieu de cela, les spécifications SOAP décrivent des aspects importants des contenus de données et des structures dans leurs relations avec des modèles de programmation familiers tels que les appels de procédures distants (*remote procedure calls* ou RPC) et les systèmes de transfert de messages.

Ces spécifications font tout simplement partie du monde XML. SOAP n'est pas lié à un langage de programmation spécifique, une plate-forme informatique ou un environnement de développement. Il existe des implémentations de SOAP fournissant des liens vers une grande variété de langages de programmation tels que C#, Perl et Java™. Sans ces implémentations, SOAP resterait une pure abstraction : un très beau concept sans existence concrète. Ce sont ces liens vers les langages de développement de logiciels qui donnent vie à SOAP ; ce qui sera le sujet de ce livre. Java se pose en candidat naturel pour le traitement XML et cela le rend parfait pour l'élaboration de services SOAP et d'applications clientes. Si vous cherchez comment développer en Java des logiciels intégrant SOAP, ce livre est exactement celui qu'il vous faut pour vous lancer.

Audience

Ce livre est destiné à tous ceux et toutes celles qui s'intéressent à la manière d'accéder avec Java à des services web de type SOAP, ainsi qu'à la façon de développer, également en Java, des services SOAP. Il a été écrit pour des programmeurs, des étudiants et des professionnels déjà familiarisés avec Java et ne couvrira donc pas les concepts de base ou la syntaxe de ce langage. Si Java ne vous est pas familier, vous devrez probablement garder à portée de main un manuel de ce langage, tel que *Introduction à Java* ou *Java in a Nutshell*, édités par O'Reilly.

Un moment dans le temps

Les spécifications SOAP sont toujours en cours d'évolution. Ce livre décrit SOAP tel que défini par la version 1.1 de ses spécifications. Bien que les concepts et les techniques cou-

verts garderont probablement leur pertinence dans les futures versions de SOAP, celles-ci comporteront certainement d'importants ajouts au fur et à mesure de la finalisation des spécifications. Les implémentations Java que nous allons examiner continueront également à évoluer. Naturellement, les descriptions et les exemples de ce livre finiront par dater et même devenir obsolètes dans un avenir probablement pas si éloigné que cela si l'on considère la vitesse à laquelle les services web évoluent. En fait, l'avenir est déjà écrit : Apache SOAP Version 2, sur lequel la plupart des exemples sont basés, est destiné à être remplacé par Apache SOAP 3 (également connu sous le nom de Axis), qui est déjà disponible sous forme de pré-version et que nous évoquerons brièvement au *chapitre 9*. Axis à son tour s'engage à supporter les spécifications des API JAX RPC et JAXM, qui elles-mêmes sont toujours en cours de développement. Une pré-version de l'implémentation de référence pour ces spécifications est disponible chez Sun Microsystems (et abordée dans le *chapitre 11*), cette version étant plus récente que la plus récente des versions d'Axis. De plus, il serait absurde de penser que les spécifications du pack JAX marqueront la fin de ce processus d'évolution. Cependant lorsque l'inévitable surviendra, vous serez armé des connaissances et de la compréhension nécessaires pour suivre les changements.

Contenu

Les chapitres de ce livre sont organisés de façon à ce que chacun d'entre eux soit bâti sur l'information présentée dans les précédents et il est donc préférable de les lire dans l'ordre.

Chapitre 1, *Introduction*

Ce chapitre fournit une vue générale de SOAP, des technologies qui lui sont liées, des problèmes inhérants à cette technologie ainsi que des comparaisons avec d'autres solutions. Il présente également Apache SOAP et GLUE, les implémentations utilisées tout au long de ce livre.

Chapitre 2, *Messages SOAP*

Ce chapitre décrit l'enveloppe SOAP, un document XML structuré contenant la « charge utile » d'une transaction SOAP entre un client et un serveur. Il couvre tous les aspects de l'enveloppe SOAP y compris les en-têtes, les éléments du corps SOAP et les fautes. Quelques détails concernant la mise en œuvre de SOAP sur HTTP sont également inclus.

Chapitre 3, *Encodage des données SOAP*

Ce chapitre couvre l'encodage de données d'une transaction SOAP, y compris les règles d'encodage et de sérialisation des éléments des données. Il commence par décrire les espaces de noms puis explore la sérialisation des types de données, tant simples que complexes.

Chapitre 4, *Services de type RPC*

Ce chapitre aborde de manière approfondie les services de type RPC basés sur SOAP. Nous y couvrirons largement les méthodes et paramètres de ces services ainsi que les détails de leur déploiement et des mécanismes permettant leur activation.

Chapitre 5, *Utilisation des types de données complexes*

Ce chapitre traite de la création de services comportant des paramètres de méthode et des valeurs de retour complexes tels que des tableaux et des Java Beans. Il couvre les mécanismes existant pour faire correspondre ces types à des classes Java, aussi bien côté client que serveur.

Chapitre 6, *Sérialisation personnalisée*

Ce chapitre couvre l'utilisation de types de données non standard, reprenant le sujet là où le *chapitre 5* l'avait laissé. Il examine certains outils et interfaces de programmation utilisés pour transmettre ou récupérer des instances de types de données personnalisés comme paramètres et valeurs de retour. Il détaille également les techniques d'écriture de classes Java pour la sérialisation et la désérialisation de types personnalisés.

Chapitre 7, *Fautes et exceptions*

Ce chapitre décrit les fautes SOAP dans leur rapport avec les exceptions Java. Il examine les mécanismes fournis par défaut ainsi que les techniques de génération et d'extension du contenu des fautes.

Chapitre 8, *Autres techniques*

Ce chapitre commence par décrire l'utilisation des services SOAP de type messagerie comme alternative au modèle RPC. Il examine également la transmission de documents XML au sein d'une enveloppe SOAP et se termine par les attachements SOAP.

Chapitre 9, *Interopérabilité et WSDL*

Ce chapitre aborde la façon de faire collaborer des clients et des serveurs développés à l'aide de technologies différentes. Une introduction à WSDL (*Web Services Description Language*) est fournie. Des exemples sont développés, couvrant l'élaboration de clients et de services à l'aide d'Apache SOAP et GLUE, une brève incursion dans Apache Axis ainsi que l'accès par des clients Java aux services Microsoft .NET.

Chapitre 10, *En-têtes SOAP*

Ce chapitre examine l'utilisation d'en-têtes SOAP qui fournissent le moyen de transmettre entre clients et services, des données situées hors de la portée du corps SOAP. Il traite du développement d'un service intermédiaire qui agit comme un routeur de messages vers un autre service. Quelques classes Java sont développées pour étendre le cadre Apache SOAP afin d'utiliser les en-têtes SOAP.

Chapitre 11, *JAX-RPC et JAXM*

Ce chapitre décrit un standard émergent: l'API Java pour RPC XML (JAX-RPC). C'est une introduction à la préversion de l'implémentation de référence de Sun. Il traite aussi bien du développement d'un service que d'un client et examine également l'utilisation d'outils de développement permettant l'accès à des services décrits par WSDL. Un commentaire sur JAXM est aussi inclus en fin de chapitre.

Conventions typographiques

`Les caractères à chasse fixe` sont utilisés pour :

- Tout ce qui peut apparaître dans un programme Java, y compris les mots-clés, les opérateurs, les types de données, les constantes, les noms de méthodes, les noms de variables, les noms de classes, les noms d'interfaces et les noms de paquetages Java.
- Les lignes de commandes et les options qui devront être entrées telles quelles à l'écran.
- Les espaces de noms.

Les caractères en italiques sont utilisés pour :

- Les chemins d'accès, les noms de fichiers et les adresses internet telles que les noms de domaines et les URL. Les italiques sont également utilisées pour les noms de fichiers exécutables.

Effectuer de fines distinctions dans un livre tel que celui-ci est généralement un combat perdu d'avance. J'ai cependant tenté de distinguer les espaces de noms (chasse fixe) des URL (italiques) même s'ils semblent identiques. De même, j'ai essayé de distinguer les méthodes Java (chasse fixe et paire de parenthèse de fin) des méthodes exportées par un service SOAP (chasse fixe, pas de parenthèses de fin).

Cette icône indique une note relative au texte situé à proximité.

Cette icône indique un avertissement relatif au texte situé à proximité.

Votre avis

J'ai tenté d'être le plus précis possible dans mes descriptions et mes exemples mais les erreurs ou les omissions sont inévitables. Si vous en rencontrez ou pensez que j'ai ignoré d'importants détails ou si vous souhaitez me contacter pour toute autre raison relative à ce travail, vous pouvez le faire directement à cette adresse :

rob@mindstrm.com

Vous pouvez également adresser à l'éditeur commentaires et questions concernant ce livre :

Éditions O'Reilly
18 rue Séguier
75006 Paris

Ce livre possède sa page web sur laquelle se trouvent les errata ou d'autres informations. Vous pouvez y accéder par l'URL :

http://www.oreilly.fr/catalogue/java_soap.html

Pour faire des commentaires ou poser des questions techniques sur ce livre, envoyez un courrier électronique à :

france@oreilly.com

Pour plus d'information sur les livres, les conférences, les centres de ressources et le réseau O'Reilly, consultez le site web O'Reilly à l'URL :

http://www.oreilly.fr

Exemples

Le code des exemples de ce livre est disponible à :

http://www.mindstrm.com/javasoap

Remerciements

Mon bon ami Rinaldo DiGiorgio continue jusqu'à ce jour à m'intéresser à Java et aux technologies liées. Je ne crois pas que quiconque ait eu plus d'influence sur mes travaux Java qu'il n'en a eu. Merci Rinaldo, de me faire rester sur le bon chemin.

De nombreux remerciements doivent aller à David Askey et Anne Thomas Manes pour leur relecture du livre et leurs précieux commentaires. Ils ont contribué à trouver des erreurs et à fournir des avis qui rendent ce livre meilleur qu'il ne l'aurait été sans leur aide. Merci à Lorraine Pecorelli pour avoir relu chaque chapitre et s'être assurée que chaque mot avait un sens. Ma plus profonde gratitude va à Mike Loukides, l'éditeur de ce livre. De nombreux obstacles se sont dressés sur la route de l'accomplissement de ce projet et l'engagement de Mike ainsi que sa loyauté ont été la clé de la transformation de tous ces efforts en un livre. Un grand merci également à l'équipe de conception et de production d'O'Reilly.

Et enfin, merci à ma famille, Jessica et Carolyn, pour leur soutien. Je ne remercierai pas mes amis cette fois, ils ne me furent d'aucune aide !

1

Introduction

Dans l'histoire du développement de logiciel, les approches novatrices ont souvent ramené sur le devant de la scène des idées oubliées. À chaque fois qu'une idée est revisitée, les réussites et les échecs précédents deviennent des aides inestimables dans l'amélioration du concept et rendent son implémentation plus réussie, ou tout au moins plus utilisable. Il ne s'agit pas de réinventer perpétuellement la roue mais de revenir sur ses pas et de l'améliorer. Ceci peut souvent être le catalyseur d'idées et de technologies nouvelles qui n'auraient pas vu le jour avec l'ancienne roue.

Nous avons vu l'informatique centralisée et ses terminaux associés revenir sous le masque des serveurs d'applications et des clients légers. Nous avons vu le concept de P-Code réapparaître sous la forme de langages interprétés tels que Java ou Visual Basic. L'univers du développement logiciel semble s'étendre et se contracter comme le cosmos. Si vous attendez suffisamment longtemps, vous devriez pouvoir réutiliser votre travail d'aujourd'hui dans un avenir plus ou moins proche.

L'essentiel est que réellement, une bonne idée reste une bonne idée qu'elle soit nouvelle ou non. Seule son opportunité importe. Ainsi en va-t-il de l'informatique distribuée. Le concept n'est pas nouveau mais il est constamment revisité. La diffusion d'infrastructures et de technologies telles que l'Internet, les navigateurs web et leurs protocoles associés nous ont permis de revenir sur nos pas et de faire progresser l'état de l'informatique distribuée. Le dernier engouement en date dans cette évolution est constitué par les services web.

Les services web sont globalement des fonctions de serveurs ayant publié les mécanismes d'interface nécessaires pour accéder à leurs capacités. Ils sont implémentés à l'aide de technologies variées mais ont une chose très importante en commun : ils sont fournisseurs de services informatiques accessibles *via* un protocole standardisé. Vous pouvez par exemple rencontrer un service de cotation boursière capable de fournir le cours d'une action et les informations commerciales associées, à partir du symbole boursier de la société. Ceci est une fonction très spécialisée et c'est l'essence même des services web. Il ne fournissent pas toute la palette de possibilités que l'on trouve sur un serveur d'applications. Ils apportent de petites fonctionnalités très spécifiques dont l'utilité n'apparaît qu'une fois combinées avec d'autres services. Vous pouvez imaginer une application boursière en ligne utilisant des services web allant de la cotation boursière aux

transactions bancaires en passant par l'exécution des ordres. Les services web montrent qu'il est au final possible de créer des applications complexes à la volée, ou tout au moins au prix d'un temps de développement minime, en combinant des fragments de données et de services distribués à travers le Web. Le slogan de Sun a été « Le réseau est l'ordinateur » et cette vision est certainement en train de devenir réalité.

RPC et systèmes distribués orientés messages

Les systèmes distribués sont pour la plupart, des entités à couplage lâche communiquant entre elles pour accomplir certaines tâches. L'un des modèles les plus courants en informatique distribuée est constitué par les appels de procédures distants (RPC, *remote procedure call*). L'une des raisons de la popularité des systèmes RPC est qu'ils sont très proches de la syntaxe et de la sémantique des appels de fonctions et de méthodes avec lesquels sont familiers les développeurs. Les technologies comme Java RMI, COM de Microsoft et CORBA utilisent toutes ce genre de modèle. Il faut naturellement surmonter quelques obstacles avant d'effectuer avec naturel un appel de méthode vers un système distant mais cela reste malgré tout très proche d'un appel local de méthode. Une fois obtenu le retour de l'appel, on se préoccupe en général peu de la façon dont cela s'est déroulé.[1] Le travail permettant d'apporter cette abstraction au programmeur au niveau de l'API est ce qui constitue l'essentiel dans l'implémentation des systèmes distribués.

Un autre modèle populaire dans l'informatique distribuée est l'envoi de messages. Contrairement au modèle RPC, l'envoi de messages n'émule pas la syntaxe des appels de fonctions des langages de programmation. Au lieu de cela, des messages composés de données structurées sont transmis entre les parties. Ces messages peuvent en quelque sorte servir à découpler les nœuds d'un système distribué et les systèmes orientés messages ont souvent prouvé une plus grande flexibilité que les systèmes basés sur le modèle RPC. Cette flexibilité toutefois peut parfois être précisément la corde avec laquelle les programmeurs se pendront.

Il semble qu'un compromis à la fois raisonnable et puissant serait de combiner ces deux modèles. Imaginez un système apportant à la fois la facilité et la familiarité des invocations de type RPC et une partie de la flexibilité des systèmes de type messages. Il me semble que cela nécessiterait la définition d'un format de données autodescriptif et se conformant à un ensemble de règles standard régissant cette description. Hmmm...

Données autodescriptives

En termes de programmation, l'expression *données autodescriptives* est elle-même autodescriptive. Autrement dit, à la question « Qu'est-ce qu'une donnée autodescriptive ? » la réponse est « Une donnée qui se décrit elle-même. » Définition fort peu utile. Mais elle est le résultat de la conception d'un format de données flexible utilisable par le plus grand nombre.

---

1. Je ne dis pas qu'il faut oublier que nous sommes en train d'effectuer des appels à des systèmes distants. Imaginez, par exemple, les conséquences d'itérations au sein d'un tableau distant composé de dix mille objets, appelant une méthode distante pour chaque cellule du tableau. Ce ne serait pas vraiment efficace !

Examinons un exemple très simple. Supposons que nous concevions un système distribué de type messages fournissant des cotations boursières. Nous pouvons concevoir le format du message de réponse comme ceci :

- les 10 premiers caractères contiennent le symbole boursier, complété à droite par des espaces ;
- les 10 caractères suivants représentent le dernier cours, complété à droite par des espaces ;
- les 10 caractères suivants représentent le volume des échanges, complété à droite par des espaces ;
- les 20 caractères suivants représentent les informations de date et d'heure de la cotation ;
- les 10 caractères suivants représentent le prix offert ;
- les 10 caractères suivants représentent le prix demandé.

Vous y êtes. Ceci est un message de format fixe. Il ne se décrit pas lui-même. Au lieu de cela, il est décrit par les spécifications données dans la liste ci-dessus. Ici, il n'y a aucune flexibilité ; et parfois il n'y a aucun besoin de flexibilité, ceci n'est pas la panacée. Mais n'admettrez-vous pas qu'il reste une marge d'amélioration ? Considérons la possibilité que les valeurs du dernier cours, du prix offert et du prix demandé puissent être mises en forme de deux façons différentes. Le premier format utilisera la notation décimale standard, 25,5 par exemple. Le second format utilisera des fractions et la même valeur sera donc représentée 25 1/2. Un format autodescriptif posséderait un sorte de clause indiquant quel format est utilisé pour chacun des champs de prix.

Prenez maintenant le même concept et appliquez-le à l'ensemble de la structure du message ainsi qu'aux parties qui le constituent. Cela vous donne la flexibilité nécessaire pour décrire pleinement le contenu d'un message. Vous pourriez par exemple rendre facultatifs certains champs de la réponse. Peut-être allez-vous demander le cours après la fermeture des marchés rendant ainsi inutiles les valeurs des prix offert et demandé. Pourquoi les renvoyer dans ce cas ? Un format autodescriptif peut spécifier son contenu et éviter de cette façon de renvoyer des données inutiles.

Pour que les données autodescriptives soient réellement utiles, chacun devra utiliser le même langage pour les décrire. Je ne parle pas ici de langage de programmation mais de langage permettant d'exprimer une description. Nous avons besoin d'une nouvelle façon de décrire et mettre en forme ces messages, une nouvelle grammaire pour tout dire. Cette nouvelle grammaire imposera les règles standard définissant le format de ces messages. C'est en fait là que l'on voit réellement la valeur ajoutée des données autodescriptives : moins dans l'élimination du problème du renvoi de données inutiles que dans le fait que tout le monde parlera le même langage. Si j'écris une application boursière utilisant 11 caractères pour représenter le symbole de la compagnie, elle ne pourra communiquer avec votre serveur utilisant, lui, 10 caractères. Mais si nous échangeons des données autodescriptives, le problème disparaît partiellement : chaque fragment de données décrira ce qu'il est d'une façon standard et il y aura donc plus de chances que des logiciels provenant d'individus ou d'organisations diverses puissent communiquer.

XML

Nous avons tous eu sous les yeux une forme de données autodescriptives à chaque fois que nous avons utilisé un navigateur web. HTML est un bon exemple de format de données standard suffisamment flexible en raison de sa capacité à prévoir des éléments autodescriptifs. Par exemple, la couleur et la police de caractères à appliquer à une section particulière du texte sont décrits avec le texte lui-même. Ce type de données autodescriptives est communément appelé *langage à balises*. Le contenu est « balisé » d'instructions destinées à sa présentation. Ceci est fort commode et a bien entendu obtenu un incroyable degré d'acceptation de la part de l'industrie du logiciel. Mais HTML n'est pas assez flexible pour recevoir du contenu qui n'aurait pas été prévu par ses concepteurs. Ce n'est pas une attaque contre HTML, c'est juste la réalité. HTML n'est pas extensible.

Le langage XML (*Extensible Markup Language*) est exactement ce que nous recherchons. XML est un langage hiérarchique et balisé au même titre que HTML. Une différence importante pour nous est qu'il est totalement extensible. Il nous permet de décrire un contenu spécifique à nos propres applications, d'une façon standard et sans que les concepteurs du langage n'aient eu à anticiper ce contenu. XML me permet par exemple de créer un contenu représentant le message de réponse boursier de la section précédente. Il définit les règles que je dois suivre pour accomplir cette tâche, sans dicter de format spécifique.

Je ne consacrerai pas de temps à XML lui-même. S'il ne vous est pas familier, vous devrez remédier à cela avant d'entamer la lecture du chapitre 2. De nombreux livres ont été publiés sur XML et les sujets qui lui sont relatifs. Un bon point de départ pour une information générale est *XML in a Nutshell*, d'Elliotte Rusty Harold et W. Scott Means (O'Reilly). Pour les développeurs Java, *Java et XML* de Brett McLaughlin (O'Reilly) devrait présenter un intérêt tout particulier.

Spécifications de type API contre spécifications de type transmission

Les programmeurs Java ont l'habitude des spécifications de niveau API dans lesquelles les classes, les interfaces, les méthodes et autres sont clairement définies afin de traiter un besoin spécifique. Ces spécifications sont conçues pour être indépendantes d'une implémentation particulière et se concentrent plutôt sur les abstractions à implémenter.

Considérez les spécifications Java Message Service (JMS). Elles décrivent pleinement l'API que les applications Java peuvent utiliser pour accéder aux fonctionnalités des logiciels middleware orientés messages (*message-oriented middleware* ou MOM). La justification d'une API standard est simple : si les fournisseurs de MOM adoptent l'API, il devient plus facile pour les programmeurs de travailler avec les diverses offres de produits. En théorie, il vous devient possible de basculer d'une implémentation JMS à une autre sans impact sur votre code. En pratique, cela signifie que les fournisseurs de produits sont quelque peu pieds et poings liés et dans l'impossibilité de fournir des API alternatives qui permettraient d'améliorer les fonctionnalités ou d'augmenter les capacités de leurs produits sans sacrifier la compatibilité. Cependant, les spécifications de type API existent depuis longtemps et font l'essentiel de ce pourquoi elles ont été conçues.

Toutefois, l'approche API présente une lacune dans un domaine extrêmement important du développement logiciel : l'interopérabilité. Vous ne pouvez développer une application utilisant l'API compatible JMS d'un fournisseur A pour communiquer avec le serveur compatible JMS d'un fournisseur B. Les spécifications traitent seulement de l'API, elles n'abordent pas le format des données échangées entre les parties. Cela semble indiquer que l'interopérabilité n'est pas aussi importante que l'entente sur la syntaxe de programmation. C'est un compromis, mais ce n'est pas forcément toujours le meilleur.

À l'inverse, des spécifications de type transmission traitent exclusivement du contenu et du format des données transmises entre les parties. Plutôt que de se concentrer sur les API, elles se consacrent à la représentation utilisée pour les interactions en informatique distribuée. Ainsi, vous pouvez être à peu près certain qu'en travaillant avec des implémentations de plusieurs fournisseurs, les API seront différentes. En revanche, vous aurez une chance raisonnable que des systèmes distribués, élaborés à l'aide de produits de différents fournisseurs fonctionnent ensemble. Pour tous travaux dans le domaine des services web, les spécifications de niveau transmission seront vos alliées. Les spécifications de type API ne vous mèneraient pas très loin.

Vue d'ensemble de SOAP

L'une des plus récentes incursions dans le monde de l'informatique distribuée et d'où découlent des spécifications de type transmission est appelée SOAP (*Simple Object Access Protocol*). Il s'agit d'un protocole relativement léger, basé sur XML et conçu pour l'échange d'information dans un environnement informatique distribué. Il n'existe pas de concept de serveur central dans SOAP. Tous les nœuds peuvent être considérés comme des égaux, voire comme des pairs.

Le protocole est constitué de plusieurs parties distinctes. La première est l'*enveloppe*, utilisée pour décrire le contenu d'un message et donner quelques indications sur la façon de le traiter. La seconde partie consiste en un ensemble de règles pour l'encodage des instances des types de données spécifiques. Ceci est l'une des parties les plus critiques de SOAP : son extensibilité. La dernière partie décrit l'utilisation de l'enveloppe et des règles d'encodage de données pour la représentation des appels et réponses RPC, y compris l'utilisation de HTTP comme couche de transport sous-jacente.

L'informatique distribuée de type RPC est aujourd'hui l'usage le plus commun de SOAP en Java, mais il n'est pas le seul. Tout transport peut être utilisé pour acheminer des messages SOAP, même si HTTP est le seul décrit dans les spécifications. D'intéressantes discussions ont lieu quant à l'usage de SMTP, BEEP, JXTA et d'autres protocoles pour l'acheminement de messages SOAP.

Utiliser SOAP avec Java

SOAP diffère de RMI, CORBA et COM en ce qu'il se concentre sur le contenu, se découplant ainsi à la fois de l'implémentation et de la couche de transport. Concept intéressant pour un livre sur Java, ne trouvez-vous pas ? Après tout, l'implémentation et le transport sont vraisemblablement bâtis à l'aide de Java. Cependant SOAP ne vise en aucune façon Java ou toute autre stratégie d'implémentation.

La réalité est que SOAP est un « capaciteur », incapable d'exister par lui-même au-delà de l'abstraction des spécifications. Pour bénéficier de SOAP comme de tout autre protocole, il faut des implémentations réelles. Java est une technologie particulièrement adaptée à l'implémentation de SOAP ou la construction de services web et d'applications utilisant SOAP comme format de transmission de données.

Implémentations de SOAP

À l'heure où j'écris ce livre, il existe des douzaines d'implémentations de SOAP et de nouvelles émergent en permanence. Certaines sont implémentées en Java, d'autres non. Certaines sont gratuites, d'autres non. Inévitablement, certaines sont bonnes et d'autres non. Il serait impossible d'établir des comparaisons une à une de toutes les implémentations disponibles ni même de les couvrir toutes. D'un autre côté, il ne serait pas judicieux de se focaliser sur une seule implémentation car cela serait un parti pris qui n'est pas dans mes intentions. Un compromis raisonnable, et c'est celui que j'ai choisi, est de sélectionner deux implémentations Java de SOAP intéressantes et de les utiliser tout au long de ce livre. Cela vous donnera l'opportunité de voir différentes API et stratégies de programmation. Dans les chapitres 9 et 11, je dérogerai à cette règle et jetterai un regard rapide sur d'autres technologies importantes SOAP.

Apache SOAP

L'*Apache Software Foundation* a un projet en cours connu sous le nom de Apache SOAP. Il s'agit d'une implémentation des spécifications SOAP pouvant être hébergée par des serveurs supportant les servlets Java. Les exemples de ce livre sont basés sur Apache SOAP version 2.2, disponible à l'adresse *http://xml.apache.org/soap/index.html*. Quatre facteurs importants m'ont conduit à choisir cette implémentation : elle supporte une grande partie des spécifications, elle possède une base d'utilisateurs raisonnablement large, le code source est disponible et elle est gratuite.

Bien qu'Apache SOAP puisse être hébergée par plusieurs serveurs, j'ai choisi Apache Tomcat (Version 3.3 et Version 4), disponible à l'adresse *http://jakarta.apache.org/tomcat/index.html*. Les raisons ne sont pas particulièrement liées à SOAP mais celui-ci fonctionne bien avec Tomcat. Son utilisation n'a pas de réel impact sur les exemples de ce livre et vous pourrez donc choisir un autre serveur si vous le souhaitez.

Apache SOAP a été développée alors qu'il n'existait pas de standards pour l'API SOAP. Les spécifications SOAP ne visent pas les liens avec les langages et les implémenteurs ont donc été obligés de développer leur propre API. Plus récemment, la communauté Java a travaillé sur des standards pour les messages basés SOAP et les API RPC, connus respectivement sous le nom de JAXM et JAX-RPC. Nous jetterons un regard à ces API au chapitre 11. Elles seront implémentées par Axis, qui est l'implémentation de la prochaine génération des serveurs Apache SOAP. Dans un monde idéal, JAXM et JAX-RPC auraient été terminés à temps, me donnant ainsi la possibilité de leur donner la place qu'ils méritent, dans la mesure où ils remplaceront certainement les API développées pour Apache SOAP 2.2. En réalité, les API standard sont tout juste en phase de consolidation et devraient apparaître sous leur forme finale juste à temps pour me faire écrire une seconde édition plus tôt que ne l'aurais voulu. La plus grosse partie de ce livre sera consacrée aux technologies utilisables pour le développement d'un système en production. Une fois les concepts acquis, la migration vers une autre API ne sera pas difficile.

GLUE

GLUE est une implémentation de SOAP développée par la société The Mind Electric (*www.themindelectric.com*). Elle est intégralement développée en Java et peut être hébergée par de nombreux serveurs supportant les servlets ou peut être exécutée seule grâce à son propre serveur HTTP. Les exemples GLUE de ce livre sont basés sur la version 2, disponible à l'adresse *http://www.themindelectric.com/products/glue/glue.html*. J'ai choisi GLUE pour quatre raisons: elle utilise une approche très orientée programmeur, ses API sont assez différentes de celles d'Apache SOAP, elle s'appuie sur le *Web Services Description Language* (WSDL) et elle est gratuite pour la plupart des utilisations.[2]

Les API GLUE sont également propriétaires. J'attends d'une future version de GLUE qu'elle adopte des standards comme JAXM et JAX-RPC si la communauté des utilisateur le réclame, mais c'est encore à voir.

Autres

Au chapitre 9 nous travaillerons un peu avec quelques autres technologies. .NET de Microsoft est un acteur majeur dans le domaine des services SOAP, ainsi nous verrons ce qu'il faut pour écrire des applications Java utilisant SOAP pour communiquer avec des services .NET.

La fondation Apache travaille actuellement sur une implémentation SOAP nouvelle génération nommée Axis. Bien qu'il soit encore un peu tôt pour couvrir celle-ci en détail, il ne fait pas de doute qu'elle remplacera Apache SOAP version 2.X. Nous nous aventurerons à écrire une application Axis simple à l'aide de la version actuelle. (Et comme je l'ai déjà indiqué, le chapitre 11 abordera les standards JAXM et JAX-RPC, qu'Axis devrait finalement implémenter.)

Notre approche

Ce livre ne couvrira certainement pas tous les aspects de la technologie SOAP. Mon but est de vous apporter une bonne compréhension des aspects majeurs de SOAP dans un contexte de développement Java. Vous verrez que beaucoup d'exemples sont présentés non seulement en Java, mais également en XML SOAP généré à l'exécution de code Java. Ceci vous donnera une idée de ce que font réellement les différentes API. Apprendre SOAP de cette façon vous mènera au-delà de l'objectif de ce livre et vous incitera à explorer les fonctionnalités des implémentations que j'ai traité ainsi que des autres.

La sécurité

Un domaine particulier ne sera pas abordé dans ce livre : la sécurité. Mais comment parler d'informatique distribuée sans aborder ce thème ? La réponse est en réalité très simple: les spécifications SOAP n'en parlent pas. Les implémentations actuelles s'appuient sur les fonctionnalités de sécurité de la technologie hébergeante. Qu'il s'agisse de SSL, de l'authentification HTTP élémentaire, de l'authentification par proxy ou de tout autre mécanisme, la sécurité dépend de la technologie accueillante et non de SOAP lui-même. On attend d'une future version des spécifications SOAP ou de spécifications

2. Voir les termes de la licence GLUE.

séparées dédiées à la sécurité qu'elles abordent ce thème, mais pour l'instant vous devrez vous appuyer sur ce que fournit la technologie qui vous hébergera.

Les spécifications actuelles mentionnent toutefois l'utilisation d'en-têtes SOAP dans des mécanismes de sécurité et vous vous rendrez compte que certaines implémentations suivent ces directives. Néanmoins, les mécanismes sont susceptibles d'être spécifiques à chaque implémentation jusqu'à l'adoption d'un standard.

Comment commencer

Si vous envisagez de travailler avec les exemples, vous devrez installer les composants logiciels nécessaires, y compris le JDK, les implémentations SOAP décrites plus haut et quelques autres technologies. Tous ces composants sont assez bien documentés et vous ne devriez pas avoir de problèmes pour les installer correctement.

Les chapitres de ce livre sont organisés de façon à ce que chacun s'appuie sur des concepts vus dans le précédent. Je suggère donc que vous les abordiez dans l'ordre, mais naturellement cela dépend de vous. Si vous êtes déjà à l'aise avec XML ou (mieux encore) avec certains aspects des services web, vous pourrez peut-être sauter quelques passages.

2

Messages SOAP

Tout message SOAP est empaqueté dans un document XML appelé une *enveloppe*, qui est un contenant structuré portant un message SOAP. La métaphore est appropriée car vous glissez dans une enveloppe tout ce dont vous avez besoin pour effectuer une opération et vous l'envoyez à un destinataire qui l'ouvrira et reconstituera le contenu original afin d'effectuer l'opération demandée. Le contenu d'une enveloppe SOAP est conforme aux spécifications SOAP,[1] permettant à l'expéditeur et au destinataire d'échanger des messages de façon indépendante du langage utilisé : l'expéditeur peut par exemple être écrit en Python et le destinataire en Java ou C#. Aucune des deux parties n'a à se préoccuper de la façon dont l'autre est implémentée car elles sont toutes deux d'accord sur la façon d'interpréter l'enveloppe. Nous allons maintenant étudier cette enveloppe SOAP.

La liaison HTTP

Les spécifications SOAP exigent d'une implémentation qu'elle supporte le transport de la charge utile XML SOAP *via* HTTP et ce n'est donc pas une coïncidence si toutes les implémentations SOAP supportent HTTP. Les implémentations sont naturellement libres de supporter d'autres couches de transport même non décrites par les spécifications. Rien n'empêche que la charge utile des messages SOAP soit transportée *via* SMTP, FTP, JMS ou même des protocoles propriétaires. De fait, les couches de transport alternatives font l'objet de discussions fréquentes et quelques unes ont été implémentées. Toutefois, HTTP étant la couche prédominante, c'est sur celle-ci que nous allons nous concentrer. Une fois que vous maîtriserez la façon dont SOAP est lié à HTTP, vous devriez pouvoir facilement transposer vos connaissances aux autres mécanismes de transport.

SOAP peut aussi bien être utilisé pour des échanges de messages de type requête/réponse comme l'est RPC, que pour des échanges à sens unique comme l'est SMTP. La majorité des implémentations SOAP récentes basées sur Java a implémenté les messages de style RPC et c'est donc sur ce type que nous nous focaliserons. HTTP est un choix natu-

1. Les spécifications sont disponibles sur *http://www.w3.org/TR/SOAP*. Les spécifications 1.1 ne sont pas une recommandation du W3C, mais SOAP 1.2 en cours de développement le sera.

rel dans les échanges de type RPC car il permet à la requête et à la réponse de constituer des parties intégrantes d'une seule et même transaction. Toutefois, les échanges à sens unique ne doivent pas être négligés et rien dans HTTP n'empêche ce type de requêtes. Nous examinerons les messages à sens unique dans le chapitre 8.

Les requêtes HTTP

Le premier message SOAP que nous allons examiner est une requête RPC. Bien que fort simple, elle contient tous les éléments requis pour être totalement compatible avec les messages SOAP utilisant HTTP. La charge utile XML du message est contenue dans une requête HTTP POST. Jetez-y un œil mais ne vous attardez pas encore sur les détails. Le message qui suit demande au serveur de renvoyer la température de son emplacement géographique en degrés Celsius :

```
POST /MeteoLocale HTTP/1.0
Host: www.mindstrm.com
Content-Type: text/xml; charset="iso-8859-1"
Content-Length: 344
SOAPAction: "StationMeteo"

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
    <m:DonneTemperatureCourante xmlns:m="StationMeteo">
         <m:echelle>Celsius</m:echelle>
    </m:DonneTemperatureCourante>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

La requête HTTP SOAP utilise la méthode HTTP POST. Bien que la charge utile SOAP puisse être transportée à l'aide d'une autre méthode comme par exemple HTTP GET, la liaison HTTP définie dans les spécifications SOAP requiert l'usage de la méthode POST. La méthode spécifie également le nom du service auquel on veut accéder. Dans cet exemple, nous envoyons les données à `/MeteoLocale` sur l'hôte spécifié plus loin dans l'en-tête HTTP. Cela indique au serveur comment router la requête en interne. Enfin, notre exemple montre que nous utilisons la version HTTP 1.0 bien que SOAP ne requiert pas de version particulière.

Le champs d'en-tête `Host:` spécifie l'adresse du serveur auquel nous envoyons cette requête, *www.mindstrm.com*. Le champ d'en-tête suivant, `Content-Type:`, indique au serveur que nous envoyons des données de type `text/xml`. Tous les messages SOAP doivent être envoyés sous forme de `text/xml`. Le type de contenu de l'exemple spécifie également que les données sont encodées à l'aide du jeu de caractères UTF-8. Le standard SOAP n'impose aucun encodage particulier. `Content-Length:` indique au serveur le nombre de caractères que contiennent les données de la charge utile XML SOAP POSTée qui suit.

Pour l'instant, les en-têtes étaient des en-têtes HTTP standard pouvant s'appliquer à n'importe quel message HTTP POST. Le suivant est toutefois spécifique à SOAP. Le champ d'en-tête `SOAPAction:` est requis pour tout message de requête SOAP transporté

par HTTP.[2] Il fournit au serveur HTTP quelques informations sous la forme d'un URI indiquant le but du message. Cette information est contenue dans un en-tête HTTP plutôt que dans le contenu du message lui-même car il ne nécessite pas le traitement préalable par le système de la charge utile XML. Cela permet au serveur de déterminer s'il possède les informations ou les ressources nécessaires pour traiter la requête, sans devoir réellement analyser le message. Bien qu'il puisse contenir des données au format quelconque, voire rester vide, ce champ doit être présent. L'en-tête `SOAPAction: "StationMeteo"` peut indiquer que notre requête requiert une connexion active à la station située sur le toit de l'immeuble dans lequel se trouve le serveur. Si le serveur sait que la station est tombée de l'immeuble et a été broyée par une voiture, il pourra répondre sans perdre de temps à traiter la charge utile SOAP. Ceci n'est peut-être pas un scénario très courant, mais l'important est que le serveur peut utiliser l'URI spécifié dans le champ `SOAPAction:` pour prendre connaissance de l'intention du message et agir en conséquence. Il est également important de savoir que l'URI n'a pas besoin d'être sous une forme particulière. Ce peut être une URL, un nom, un mot ou même un nombre tant que cela présente une signification pour le serveur recevant le message.

Si le champ `SOAPAction:` contient une chaîne vide (`""`), l'intention du message sera alors fournie par l'URI de la requête HTTP qui dans notre exemple est `/MeteoLocale`. Le serveur peut interpréter cet URI comme une demande d'accès à la station météo ou elle peut avoir une autre signification. Si le champ ne contient pas de données, le message ne contiendra alors aucune indication quant à sa signification ou son intention. Dans ce cas, nous attendrons du serveur qu'il continue et traite la charge utile XML.

La réponse HTTP

Les messages de type requête RPC amènent en général une réponse HTTP. Naturellement, si le serveur ne peut obtenir l'information dont il a besoin dans les en-têtes HTTP, il peut répondre par une erreur HTTP particulière. Mais en supposant que les en-têtes ont été correctement interprétées, on attend du système qu'il apporte une réponse SOAP. Voici la réponse HTTP à la requête RPC de l'exemple précédent:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset="iso-8859-1"
Content-Length: 370

<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
     <m:DonneTemperatureCouranteResponse xmlns:m="StationMeteo">
          <m:temperature>26,6</m:temperature>
     </m:DonneTemperatureCouranteResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Les champs d'en-tête HTTP de la réponse sont en grande partie similaires à ceux de la requête. Le code réponse 200 de la première ligne de l'en-tête indique que le serveur a

2. Ceci est obligatoire en SOAP 1.1 mais devrait être facultatif en SOAP 1.2.

été en mesure de traiter la charge utile XML SOAP. Les champs `Content-Type:` et `Content-Length:` ont la même signification que dans le message de requête. Aucun autre champ d'en-tête HTTP n'est nécessaire. La corrélation entre la requête et la réponse est impliquée par le fait que le mécanisme requête/réponse est inhérent au POST HTTP. Vous envoyez une requête et recevez une réponse au sein d'une seule et même transaction.

Modifions le message original de la requête de façon à ce que l'échelle soit :

```
<m:echelle>Calcium</m:echelle>
```

Vous connaissez cette échelle n'est-ce pas ? Et bien non, il n'existe pas d'échelle Calcium pour la température. Nous avons fabriqué une requête erronée. Continuons et envoyons la requête SOAP au serveur. En supposant que la station météo n'est pas tombée de l'immeuble, le serveur va traiter la requête. Comme nous pouvions nous y attendre, notre code SOAP ne comprend pas ce qu'est l'échelle de température Calcium et répond par l'erreur suivante :

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="iso-8859-1"
Content-Length: 525

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:Client</faultcode>
         <faultstring>Erreur Client</faultstring>
         <detail>
            <m:detailsfautemeteo xmlns:m="StationMeteo">
               <message>Pas d'echelle de tempérarure : Calcium</message>
               <errorcode>1234</errorcode>
            </m:detailsfautemeteo>
         </detail>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Le code erreur HTTP 500 et son message associé « Internal Server Error » montrent qu'une erreur est survenue. Comme précédemment, le serveur inclut les champs d'en-tête `Content-Type:` et `Content-Length:`. Les spécifications SOAP précisent que si le message et reçu et compris, la réponse doit être envoyée avec un code de statut 2xx. Si le serveur ne comprend pas le message, si le message est incorrectement mis en forme, qu'une information en est absente ou qu'il ne peut être traité pour une autre raison, le serveur doit utiliser le code HTTP 500 (« Internal Server Error », « Erreur Interne du Serveur »). Cet en-tête HTTP est suivi d'une enveloppe SOAP contenant son code de faute, le texte de la faute et une description détaillée de l'erreur.

Je ne suis pas convaincu que ceci soit le meilleur moyen de traiter les fautes SOAP ni que l'utilisation d'une erreur du protocole de transport soit une bonne façon de présenter une erreur SOAP. Les erreurs de la couche de transport et les erreurs SOAP n'ont absolument rien en commun. La réponse inclut, après tout, un message SOAP adéquat décrivant l'erreur en détail. Cependant, par équité envers les auteurs des spécifications

SOAP, il faut reconnaître que cette procédure de traitement d'erreur est conforme à la façon dont HTTP renvoie ses erreurs lorsqu'il accède à des documents HTML. Je pourrais continuer à argumenter dans un sens ou dans l'autre mais ceci est défini ainsi et n'interfère certainement pas avec le traitement SOAP.

Résumé de la liaison HTTP

Ceci est tout ce dont nous avons besoin pour traiter des liaisons SOAP HTTP. Toute la beauté de la chose réside dans le fait que HTTP est vraiment un mécanisme bien établi et très bien adapté au transport de XML et donc une bonne façon de transporter des messages SOAP.

Certains détails du XML des exemples précédents peuvent vous sembler évidents même si nous ne les avons pas encore abordés. Si vous comprenez le XML dans le texte, parfait. Dans le cas contraire, ne vous inquiétez pas. Le but de ce que nous avons vu jusqu'à présent était de vous montrer à quoi ressemblait la liaison HTTP SOAP. Dans la section suivante, nous examinerons de façon détaillée l'enveloppe SOAP qui constitue la charge utile contenant les requêtes et les réponses SOAP.

L'enveloppe SOAP

L'enveloppe SOAP représente l'intégralité du XML d'une requête ou d'une réponse SOAP.[3] L'`envelope` est l'élément XML de niveau le plus élevé dans le message et doit être présente pour que le message soit considéré comme valide. Par essence, l'`envelope` représente donc le document XML contenant le message. L'`envelope` peut contenir un élément `Header` facultatif qui, s'il est présent, doit être le premier sous-élément de l'`envelope`. L'`envelope` doit contenir un élément `Body`.[4] Si l'`envelope` contient un élément `Header`, l'élément `Body` doit alors être placé immédiatement après le `Header`. Sinon, le `Body` doit être le premier sous-élément de l'`envelope`.

Une enveloppe SOAP empaquetée et transportée à l'aide de HTTP est similaire à une enveloppe papier envoyée par la Poste. L'enveloppe SOAP constitue le papier de l'enveloppe, l'en-tête SOAP (si présent) et le corps SOAP sont le contenu de l'enveloppe papier et les en-têtes HTTP sont les informations d'adresse physique situées à l'extérieur de l'enveloppe papier.

Avec un brin d'imagination, nous pouvons pousser l'analogie. L'adresse de retour d'une enveloppe papier n'a pas de contrepartie directe dans le processus SOAP. Toutefois, dans les deux cas, un message non-délivrable sera renvoyé à l'expéditeur. La seule chose manquante est le timbre. C'est je crois là que l'analogie s'arrête. Et encore, bien que vous n'ayez pas à payer pour envoyer une requête HTTP, vous le faites probablement pour votre accès à l'Internet. Voila donc votre affranchissement postal !

Espaces de noms

Comprendre les espaces de noms XML est important pour comprendre SOAP. Je ne couvrirai pas tous les détails des espaces de noms XML car cela prendrait trop de temps et

3. Une exception survient lorsqu'un attachement est inclus. Nous examinerons cette possibilité au chapitre 8.
4. Des discussions sont en cours sur la possibilité de fournir des enveloppes sans `Body` en SOAP 1.2.

déborderait du sujet de ce livre. Au lieu de cela, je vais vous en donner une description rapide dans leur relation avec SOAP. De façon élémentaire, les espaces de noms sont un mécanisme XML permettant d'éliminer l'ambiguïté entre des éléments ou attributs XML. En d'autres termes, ils nous aident à comprendre le contexte ou la signification d'un élément. Revenons à notre exemple précédent :

```
<m:DonneTemperatureCourante xmlns:m="StationMeteo">
      <m:echelle>Celsius</m:echelle>
</m:DonneTemperatureCourante>
```

XML nous permet de créer quelque chose de similaire aux variables d'un langage de programmation pour représenter les espaces de noms. Nous avons créé dans l'exemple ci-dessus, un identifiant d'espace de noms nommé `m` en utilisant la déclaration d'espace de noms `xmlns:m="StationMeteo"`. Généralement, la syntaxe d'une déclaration d'espace de noms dans un message SOAP est :

```
xmlns:id="un URI"
```

Le mot-clé `xmlns` indique au processeur XML que nous définissons un espace de noms. Si la définition comprend un nom de qualificateur, c'est que nous sommes en train de créer un qualificateur d'espace de noms qui peut être utilisé pour distinguer cet espace de noms des autres. Si nous ne spécifions pas de nom de qualificateur, nous définissons un espace de noms par défaut. L'`id` est le nom d'identifiant que nous déclarons, dans notre cas la lettre `m`. La dernière partie est une URI fournissant le contexte pour l'espace de noms. Bien que l'URI soit fréquemment une URL, il est important de comprendre qu'il ne s'agit que d'un identifiant. Le destinataire du document n'essaiera pas de charger le contenu de l'URI (qui peut fort bien ne même pas exister). Nous pouvons utiliser la lettre `m` pour qualifier les éléments situés dans la portée de la déclaration elle-même.

Qualifier un élément XML signifie l'associer à un espace de noms spécifique. Dans cet exemple, nous qualifions l'élément XML `DonneTemperatureCourante` à l'aide de l'espace de noms `StationMeteo` en déclarant l'élément comme étant `m:DonneTemperatureCourante`. Ceci signifie que `DonneTemperatureCourante` est associé à l'espace de noms `StationMeteo` représenté par l'identifiant `m`.

Notez que l'identifiant d'espace de noms `m` peut être utilisé pour qualifier l'élément dans lequel il est déclaré. Mais il n'y a rien de particulier dans le fait d'utiliser `m` dans cet élément. Nous ne sommes pas obligés de qualifier l'élément `DonneTemperatureCourante` à l'aide de l'identifiant d'espace de noms `m` parce que cet élément contient la déclaration d'espace de noms comme attribut, mais plutôt parce que nous avons besoin d'indiquer où `DonneTemperatureCourante` est défini. Placer la déclaration dans cet élément, fixe la portée de l'identifiant `m`. Il est clairement valide d'utiliser l'identifiant au même niveau que sa déclaration. C'est ce que nous avons fait dans l'exemple. Ainsi la portée de l'espace de noms est fixée par l'élément contenant la déclaration. Cela signifie que l'ID d'espace de noms peut être utilisé sur l'élément contenant la déclaration, n'importe quel attribut de cet élément et n'importe quels sous-éléments et attributs associés de l'élément contenant. L'identifiant d'espace de noms n'est pas dans la portée des éléments XML situés à un niveau supérieur de la hiérarchie. Dans l'exemple original, cela signifie que l'identifiant d'espace de noms `m` ne peut être utilisé pour les attributs des éléments du `Body` car `m` est hors de portée. Les règles de portée sont similaires à celles utilisées dans les langages de programmation de type blocs comme Java dans lesquels les variables sont envoyées dans une pile lorsque l'on entre dans des blocs de code déli-

mités par des accolades, ladite pile étant vidée à la sortie du bloc. Voici un petit morceau de code Java montrant la même notion de portée pour les variables `var1` et `var2` :

```
int var1 = 10;
{
    int var2 = 2 * var1; // OK car var1 est dans la portée
}
var1 = var2; // invalide car var2 est en dehors de la portée
```

XML supporte ce que l'on appelle un *espace de noms par défaut*, qui permet que la qualification par espace de noms soit héritée sans que ceci ne soit explicitement indiqué. Un espace de noms par défaut est déclaré en assignant une valeur à l'attribut nommé `xmlns` sans utiliser d'identifiant d'espace de noms associé. Considérez l'exemple suivant :

```
<DonneTemperatureCourante xmlns="StationMeteo">
    <echelle>Celsius</echelle>
</DonneTemperatureCourante>
```

Dans ce cas, les éléments `DonneTemperatureCourante` et `echelle` sont associés à l'espace de noms `StationMeteo`.

La qualification des espaces de noms est souvent nécessaire pour déterminer le sens que l'on veut donner à un élément ou un attribut. Considérez maintenant l'exemple suivant :

```
<controlecamion>
    <echelle>Kilogramme</echelle>
    <masse>12000</masse>
    <echelle>Celsius</echelle>
    <temperature>25</temperature>
</controlecamion>
```

Ce fragment XML contient des données qui ont pu être collectées dans une station de contrôle de camions. La première occurrence de l'élément `echelle` spécifie l'échelle utilisée pour peser le camion. Un peu après, nous avons le même élément `echelle`, indiquant l'échelle de température utilisée pour mesurer la température extérieure du lieu où est effectué le contrôle. Comment alors connaître la signification des deux éléments échelle ? La structure globale se comprend certes intuitivement, mais ce pourrait ne pas être toujours le cas.

Une manière d'éviter ce problème serait certainement d'utiliser plus de descriptifs ou des structures XML représentant mieux le sens de chaque élément. Mais ceci ne sera pas toujours contrôlable. Vous pouvez avoir à créer un document composite à partir de plusieurs fragments provenant de nombreuses sources différentes. L'utilisation de qualifications d'espaces de noms appropriées peut aider à résoudre les conflits de noms et les ambiguïtés. Le document suivant est identique au précédent mais avec les deux éléments `echelle` correctement qualifiés :

```
<controlecamion xmlns:ns1="Masse" xmlns:ns2="Temperature">
    <ns1:echelle>Kilogramme</ns1:echelle>
    <masse>12000</masse>
```

```
    <ns2:echelle>Celsius</ns2:echelle>
    <temperature>25</temperature>
</controlecamion>
```

La première occurrence d'echelle est maintenant associée à l'espace de noms Masse, alors que la seconde occurrence d'echelle est associée à l'espace de noms Temperature. Avez-vous remarqué que les deux déclarations d'espaces de noms sont des attributs du même élément ? Ce qui est parfaitement valide, les déclarations d'espaces de noms n'étant rien d'autre que des attributs et un élément XML pouvant posséder plus d'un attribut.

SOAP définit deux espaces de noms utilisables par les messages. L'enveloppe SOAP est qualifiée par l'espace de noms http://schemas.xmlsoap.org/soap/envelope. Nous avons utilisé cet espace de noms dans nos exemples précédents en déclarant l'identifiant d'espace de noms SOAP-ENV et en l'utilisant pour qualifier l'élément Envelope. L'identifiant d'espace de noms http://schemas.xmlsoap.org/soap/encoding est utilisé pour déclarer l'attribut encodingStyle, dont nous discuterons plus loin. Notez que cet attribut encodingStyle est qualifié par un espace de noms à l'aide de l'identifiant SOAP-ENV. En voici de nouveau un rapide aperçu :

```
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        . . .
        . . .
</SOAP-ENV:Envelope>
```

L'élément Envelope

L'Envelope est l'élément le plus élevé du document XML représentant le message SOAP. L'Envelope est le seul élément de niveau supérieur. Le reste du message SOAP réside dans l'Envelope, encodé sous forme de sous-éléments. Un message SOAP doit avoir un élément Envelope. L'Envelope peut contenir des déclarations d'espaces de noms comme on le voit dans les exemples précédents et doit être qualifiée, comme on l'a vu, à l'aide de l'espace de noms http://schemas.xmlsoap.org/soap/envelope. C'est pour cela que le nom d'élément est présenté sous la forme SOAP-ENV:Envelope. Il est également habituel que l'élément Envelope contienne la déclaration de l'attribut encodingStyle, également qualifié par l'identifiant déclaré d'espace de noms SOAP-ENV.

Tous les sous-éléments et attributs de l'Envelope doivent eux-mêmes être qualifiés par un espace de noms. Ces éléments et attributs sont également qualifiés par SOAP-ENV, tout comme l'Envelope l'est. Pour le reste de ce chapitre et du livre, nous utiliserons l'identifiant d'espace de noms SOAP-ENV pour signifier l'espace de noms http://schemas.xmlsoap.org/soap/envelope. Ceci devrait rendre la lecture plus facile. Gardez à l'esprit que c'est l'espace de noms lui-même qui compte et non le nom utilisé pour le qualificateur.

L'élément Header

L'en-tête SOAP est facultatif. S'il est présent, il doit être nommé Header et être le premier sous-élément de l'élément Envelope. L'élement Header est également qualifié par un espace de noms à l'aide de l'identifiant SOAP-ENV.

Plus communément, les entrées Header sont utilisées pour encoder les éléments utilisés pour le traitement des transactions, l'authentification ou d'autres informations relatives au traitement ou au routage du message. Ceci est utile car, comme nous allons le voir, l'élément Body est utilisé pour encoder l'information représentant la charge utile RPC (ou une autre). Le Header est un mécanisme d'extension permettant à tout type d'information de résider hors de la sémantique du message, le Body, mais il n'est toutefois ni utile ni même nécessaire pour le traitement du message à proprement parler.

Les éléments Header doivent restreindre l'utilisation d'attributs aux éléments descendants directs de l'élément Header lui-même. Les spécifications précisent que ceci *devrait* être le cas, signifiant ainsi que bien que l'on puisse les utiliser sur des éléments présents plus bas dans la hiérarchie située sous l'élément Header, le destinataire se doit d'ignorer les attributs de tels éléments.

Voici un exemple d'en-tête SOAP contenant un élément fils direct nommé nomutilisateur. Nous n'appliquons aucun attribut à l'élément nomutilisateur, mais nous le qualifions par un espace de noms. L'élément nomutilisateur identifie l'utilisateur à l'origine de la requête.

```
<SOAP-ENV:Header>
    <ns1:nomutilisateur xmlns:ns1="ManuelJavaSoap">
      Jessica
    </ns1:nomutilisateur>
</SOAP-ENV:Header>
```

L'attribut actor

Un message SOAP transite souvent par un ou plusieurs intermédiaires avant d'être traité. Un service proxy SOAP, par exemple, peut être situé entre une application cliente et le service SOAP visé. Nous verrons un exemple intéressant de ceci au chapitre 10, dans lequel nous développerons un service SOAP proxy, accédant à un autre service de la part d'une application cliente spécifiant l'adresse du service final. De ce fait, l'en-tête peut contenir une information destinée uniquement à l'intermédiaire ainsi qu'une information destinée au service final. L'attribut actor identifie (implicitement ou explicitement) le destinataire de l'élément Header.

Il est important de comprendre les obligations que SOAP attend d'un intermédiaire. Il demande essentiellement que tout élément SOAP Header destiné à être utilisé par un intermédiaire ne soit pas transmis au processeur SOAP suivant. Les éléments Header représentent un contrat entre l'expéditeur et le destinataire. Toutefois, si l'information contenue dans un élément Header destiné à un intermédiaire est également nécessaire à un serveur en aval, l'intermédiaire peut insérer le Header approprié dans le message avant son réacheminement. En fait, l'intermédiaire est libre d'ajouter tout élément Header qu'il juge nécessaire.

Si aucun attribut actor n'apparaît dans l'élément Header, on suppose que l'élément est destiné au destinataire final. Par essence, ceci est équivalent à l'inclusion d'un attribut actor ayant pour valeur l'URI de la destination finale.

Étendons l'exemple précédent en y ajoutant un attribut actor. Considérons que le message est envoyé à un serveur d'application intermédiaire situé à l'adresse *http://www.mindstrm.com/AppServer*. Nous voulons que ce serveur enregistre le nom de l'utili-

sateur qui effectue la requête, puis envoie celle-ci au serveur destinataire final. Pour cela, nous indiquons que l'actor pour l'élément nomutilisateur est *http://www.mindstrm.com/AppServer*:

```
<SOAP-ENV:Header>
   <ns1:nomutilisateur
         xmlns:ns1="ManuelJavaSoap"
         SOAP-ENV:actor="http://www.mindstrm.com/AppServer">
         Jessica
    </ns1:nomutilisateur>
</SOAP-ENV:Header>
```

Le serveur d'application proxy voit que l'élément nomutilisateur lui est destiné et non pas au serveur final et utilise donc les données de façon appropriée. Il retire l'élément nomutilisateur de l'en-tête avant de transmettre le message à son destinataire final. Comme vous pouvez le voir, l'attribut actor est qualifié par un espace de noms à l'aide de l'identifiant SOAP-ENV, car l'attribut actor est défini par SOAP et spécifié par l'espace de noms associé http://schemas.xmlsoap.org/soap/envelope.

L'attribut mustUnderstand

SOAP inclut le concept d'éléments d'en-tête facultatifs ou obligatoires. Ceci ne signifie pas que l'inclusion de ces éléments soit obligatoire, ce qui est un problème applicatif. Au lieu de cela, « obligatoire » signifie que le destinataire se doit de comprendre et de faire un usage approprié de l'information fournie par l'élément Header. Cette obligation nous permet de nous adapter aux situations dans lesquelles le destinataire d'un message SOAP ne peut accomplir sa tâche tant qu'il ne sait pas quoi faire des données fournies par un élément Header particulier. Dans ce cas, l'élément peut inclure un attribut mustUnderstand auquel sera assignée la valeur 1.

Ceci peut être nécessaire si l'application émettrice est mise à jour par une nouvelle version par exemple. Cette nouvelle version peut utiliser une information nouvelle devant être traitée par le serveur pour que le résultat ait une utilité. On s'attendra bien entendu à ce que le serveur ait été mis à jour de la même façon mais ceci peut ne pas encore avoir eu lieu. En raison de la discordance des versions, l'ancienne version du serveur ne comprendra pas le nouvel élément d'en-tête SOAP reçu de l'application cliente mise à jour. Supposons par exemple que l'élément d'en-tête nomutilisateur doive être compris par le destinataire. Si ce n'est pas le cas, le message devra être rejeté. Nous pouvons inclure cette obligation dans le message SOAP en assignant à l'attribut mustUnderstand la valeur 1. (La valeur 0 revient à ne pas positionner cet attribut.) Modifions notre exemple précédent afin d'indiquer que le destinataire doit comprendre l'élément nomutilisateur:

```
<SOAP-ENV:Header>
    <ns1:nomutilisateur
        xmlns:ns1="ManuelJavaSoap"
        SOAP-ENV:actor="http://www.mindstrm.com/AppServer"
        SOAP-ENV:mustUnderstand="1">
        Jessica
    </ns1:nomutilisateur>
</SOAP-ENV:Header>
```

Si le destinataire de ce message ne comprend pas l'élément nomutilisateur, il devra répondre par une faute SOAP. La réponse pourra ressembler à quelque chose comme ceci :

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="iso-8859-1"
Content-Length: 373

<SOAP-ENV:Envelope
         xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:MustUnderstand</faultcode>
         <faultstring>Erreur SOAP Must Understand</faultstring>
         <faultactor>http://www.mindstrm.com/AppServer</faultactor>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

La réponse inclut faultcode, faultstring et faultactor. faultactor indique où la faute est survenue. Nous examinerons ceci de façon détaillée plus loin dans ce chapitre.

L'attribut encodingStyle

L'attribut encodingStyle spécifie les règles d'encodage utilisées pour sérialiser et désérialiser les données. Il est important de comprendre que SOAP ne spécifie aucune règle par défaut pour la sérialisation des données. Il spécifie toutefois un mode d'arrangement simple des données et communément supporté par les implémentations. Ceci sera le sujet du chapitre suivant et nous n'entrerons donc pas dans le détail des règles d'encodage tout de suite. Il est cependant important de comprendre comment l'on spécifie le style d'encodage à utiliser pour sérialiser et désérialiser les données d'un message SOAP.

L'attribut encodingStyle est qualifié par un espace de noms à l'aide de l'identifiant SOAP-ENV. Dans l'exemple suivant, nous spécifierons l'attribut encodingStyle comme un constituant de l'élément Envelope. Nous verrons dans les chapitres suivants, des exemples effectuant cette déclaration dans l'élément Body. Les deux approches fonctionnent du moment que vous avez conscience que l'attribut encodingStyle s'applique à l'élément dans lequel il est déclaré ainsi qu'à tous ses sous-éléments (autrement dit, à toute sa portée).

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    . . .
    . . .
</SOAP-ENV:Envelope>
```

Vous pouvez fournir plus d'un URI dans la déclaration. Dans ce cas, le premier doit être le plus particulier et le dernier le moins. Voici un exemple :

```
SOAP-ENV:encodingStyle="http://www.mindstrm.com/tightEncoding
http://www.mindstrm.com/looseEncoding"
```

Dans cette situation, le système examine d'abord les règles d'encodage en utilisant *http://www.mindstrm.com/tightEncoding*.[5] Si aucune règle ne peut être trouvée avec cet URI, le système tente sa chance avec *http://www.mindstrm.com/looseEncoding*. Il est possible de désactiver le style d'encodage en cours en spécifiant une chaîne vide comme URI (""). Cette déclaration s'applique à l'élément courant ainsi qu'à tous ses sous-éléments.

Gestion des versions de l'enveloppe

Les spécifications SOAP ne définissent aucun système de numérotation permettant de déclarer quelle version d'enveloppe SOAP vous utilisez. Dans SOAP 1.1, toutes les enveloppes doivent être associées à l'espace de noms `http://schemas.xmlsoap.org/soap/envelope`,[6] que nous avons utilisé dans tous nos précédents exemples. Et rien de plus. Aucune autre information de version n'est utilisée à l'heure actuelle pour les enveloppes SOAP. Si un message SOAP est associé à un autre espace de noms ou si aucun espace de noms n'est déclaré, le destinataire devra répondre par une faute indiquant la discordance de versions. Voici un exemple de réponse dans une telle situation :

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="iso-8859-1"
Content-Length: 311

<SOAP-ENV:Envelope
         xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:VersionMismatch</faultcode>
         <faultstring>Discordance de version d'enveloppe SOAP</faultstring>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

L'élément Body

L'élément SOAP Body est obligatoire dans SOAP 1.1. S'il n'y a pas d'élément d'en-tête SOAP, l'élement Body doit être le premier sous-élément de Envelope. Sinon, il doit suivre immédiatement l'élément SOAP Header. L'élément Body est également qualifié par un espace de noms à l'aide de l'identifiant `SOAP-ENV`.

L'élément Body contient la requête ou la réponse SOAP. C'est là que l'on pourra trouver un message de type RPC contenant le nom de la méthode ainsi que ses paramètres, un message à sens unique et ses composants appropriés ou une faute et ses détails. Les éléments SOAP Body ne sont pas complètement définis par les spécifications SOAP. Nous traiterons ce sujet en détail plus tard. SOAP définit en fait une seule sorte d'élément Body : Fault SOAP.

---

5. Il est tentant d'imaginer le système essayant de télécharger `tightEncoding`, en particulier du fait que l'URI est une URL. Mais ce n'est pas le cas. Le système vérifie simplement dans ses propres tables s'il comprend `tightEncoding` et réagit de façon appropriée.
6. Pour SOAP 1.2, l'espace de noms est `http://www.w3.org/2001/12/soap-envelope`.

Revenons à l'un de nos précédents exemples d'enveloppe contenant un élément SOAP Body. La requête sera, dans ce cas, un RPC. Nous avons qualifié l'élément Body par un espace de noms, en utilisant le même identifiant SOAP-ENV que pour les autres éléments et attributs SOAP.

```
<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
         <m:DonneTemperatureCourante xmlns:m="StationMeteo">
               <m:echelle>Celsius</m:echelle>
         </m:DonneTemperatureCourante>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Les fautes SOAP

Fault est la seule entrée de l'élément Body qui soit définie par SOAP. Elle est utilisée pour transmettre les erreurs à l'initiateur d'un message SOAP. L'élément Fault doit apparaître comme un sous-élément immédiat de l'élément Body et ne peut apparaître plus d'une fois.

SOAP définit quatre sous-éléments de l'élément Fault. Tous ne sont pas obligatoires et certains ne sont appropriés que sous certaines conditions. Ces éléments sont décrits dans les sections suivantes. Comme vous allez probablement vous rendre compte que nous les avons tous déjà vu dans les exemples précédents, vous souhaiterez peut-être revenir en arrière et y jeter un œil avant d'aborder leur description.

L'élément faultcode

L'élément faultcode fournit sur la faute une indication reconnaissable par un traitement, donnant ainsi au système recevant la faute, l'opportunité d'agir en conséquence. Le code n'est pas nécessairement d'un grand secours à un humain, ceci étant le rôle de l'élément faultstring décrit dans la section suivante. L'élément faultcode est obligatoire et doit apparaître comme un sous-élément de l'élément Fault. SOAP définit de nombreux codes de fautes utilisables dans l'élément faultcode. Ces codes sont associés à l'espace de noms http://schemas.xmlsoap.org/soap/envelope. Voici une brève description des codes de fautes définis par SOAP :

`VersionMismatch`

Indique qu'un espace de noms invalide a été associé à l'élément Envelope SOAP.

`MustUnderstand`

Signifie qu'il existe un élément SOAP Header contenant un attribut mustUnderstand possédant la valeur 1 et que l'attribut n'a pas été compris ou que la sémantique qui lui a été associée ne peut être suivie.

`Client`

Indique que des erreurs sont survenues dans la mise en forme du message SOAP ou que le message ne contient pas l'information appropriée ou nécessaire. Le client doit prendre en compte le fait que le message ne pourra être renvoyé sans que les

modifications appropriées n'aient été apportées. Nous avons vu un exemple de ceci lorsque nous avons demandé la température locale en utilisant une échelle invalide.

`Server`
: Indique que le message ne peut être traité pour des raisons qui ne sont pas liées à sa mise en forme ou à son contenu. Ceci peut être interprété comme le fait que le message pourra peut-être être réexpédié sans modification ultérieurement. Le serveur de base de données nécessaire pour l'accomplissement de l'action demandée peut être à ce moment hors-ligne et sera peut-être de nouveau disponible plus tard.

Ces codes de fautes sont extensibles. Cela signifie qu'ils peuvent être étendus à l'aide de la notation point dans laquelle le nom suivant chaque point fournit un détail plus particulier. Par exemple :

```
Server.EchecStationMeteo
```

Bien que ceci puisse toujours être considéré comme une erreur `Server`, elle fournit plus d'information en indiquant qu'un problème est survenu sur l'équipement de la station météo nécessaire à l'accomplissement de la requête. Nous pouvons étendre un peu plus le code de la faute de cette façon :

```
Server.EchecStationMeteo.MiseEnPieces
```

Ce code de faute nous informe que l'équipement de la station météo est définitivement perdu, probablement en raison de sa chute et de la destruction qui s'en est suivi. Souvenez-vous toutefois que ces codes doivent être traités par un programme. Les codes de fautes étendus doivent être définis en anticipant sur les différentes situations et états possibles, permettant ainsi au destinataire d'agir de façon appropriée. (Je ne suis pas certain que la destruction d'un équipement soit une situation que l'on puisse anticiper.)

L'élément faultstring

Cet élément est également obligatoire et doit apparaître en tant que sous-élément de l'élément `Fault`. Son but est de fournir une description de la faute qui soit compréhensible par un humain. Il n'est pas destiné à être traité par le destinataire comme l'élément `faultcode` l'est. On attend de l'expéditeur qu'il garnisse cet élément d'une description ayant un sens pour le lecteur dans le contexte de la requête d'origine.

L'élément faultactor

Cet élément n'est requis que sous certaines conditions et, lorsque présent, doit être un sous-élément de l'élément `Fault`. Il est le corollaire de l'attribut d'en-tête `actor` décrit plus haut. Il fournit une indication sur le système responsable de la faute. Nous avons évoqué plus haut les intermédiaires SOAP et pris comme exemple un serveur d'application proxy. Dans cet exemple, le proxy provoquait la faute car il ne savait pas quoi faire de l'élément `nomutilisateur`. Le serveur proxy n'étant pas le destinataire final du message d'origine, il se doit d'inclure l'élément `faultactor`, s'identifiant ainsi comme la cause de la faute. La valeur de cet élément est l'URI de l'auteur de la faute.

Si la faute est causée par le destinataire final, l'élément `Fault` ne doit pas obligatoirement inclure un élément `faultactor`. Celui-ci peut toutefois être inclus.

L'élément detail

L'élément `detail` fournit une information relative aux fautes survenant en raison d'erreurs sur l'élément `Body` du message requête. Si le contenu (ou le manque de contenu) de l'élément `Body` empêche un traitement correct du message, l'élément `detail` doit alors apparaître en tant que sous-élément de `Fault`. Inversement, si la faute n'est pas liée à l'élément `Body` du message, l'élément `detail` ne doit pas être inclus. SOAP spécifie que l'absence d'un élément `detail` indique que la faute n'est pas liée au traitement de l'élément `Body`.

En particulier, l'élément `detail` ne peut être utilisé pour décrire les fautes relatives aux éléments SOAP `Header`. Dans l'exemple précédent, le serveur d'application proxy n'était pas en mesure de décrire la faute en utilisant un élément `detail` car le problème n'était pas lié au contenu de `Body`.

Autre exemple de faute

Voici un autre exemple complet de faute SOAP :

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="iso-8859-1"
Content-Length: 595

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:Client</faultcode>
         <faultstring>Erreur Client</faultstring>
         <faultactor>http://www.mindstrm.com/LocalWeather</faultactor>
         <detail>
            <m:detailsfautemeteo xmlns:m="StationMeteo">
               <message>Pas d'échelle : Calcium</message>
               <error_code>1234</error_code>
            </m:detailsfautemeteo>
         </detail>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Ceci est la même réponse que celle utilisée plus haut. La seule différence est que cette fois, nous avons inclus l'élément `faultactor` pour identifier la source de la faute. Cet exemple inclus donc une instance de chaque code de faute SOAP, ainsi qu'un élément `detail` décrivant un peu plus la faute.

À ce stade, nous en avons fini avec les messages SOAP en eux-mêmes. Notre prochaine étape consistera à examiner les encodages utilisés dans les messages.

3

Encodage des données SOAP

Lors de leur envoi à travers un réseau, les données doivent être compatibles avec le protocole de transmission sous-jacent et mises en forme de façon à ce que les parties expéditrices et réceptrices en comprennent la signification. C'est ce que nous appelons l'*encodage de données*. L'encodage de données englobe l'organisation de la structure des données, le type des données transférées et bien évidemment les valeurs des données proprement dites. Tout comme en Java, ce sont les données qui sont sérialisées et non le fonctionnement. Les règles d'encodage de données et de sérialisation aident les participants à une transaction SOAP à comprendre la signification et le contenu du message. Le modèle d'encodage SOAP est basé sur l'encodage de données XML, mais ces règles sont quelque peu altérées et contraintes afin de respecter l'objectif de SOAP. Je pense que vous vous rendrez compte que les exigences en terme de données de la plupart des systèmes peuvent aisément être représentées à l'aide des règles d'encodage que nous allons présenter.

Schémas et espaces de noms

Les espaces de noms fournissent le mécanisme utilisé pour déterminer comment les noms d'éléments et d'attributs sont interprétés. XML permettant l'usage de noms d'éléments arbitraires, il requiert un mécanisme spécifiant quel dictionnaire est à utiliser pour déterminer la signification d'un nom donné. Le style d'encodage défini dans la section 5 des spécifications SOAP est le plus utilisé. Ce style, défini dans le schéma référencé par l'espace de noms `http://schemas.xmlsoap.org/encoding`, est souvent désigné comme la « Section 5 SOAP ». Dans les messages SOAP, cet espace de noms est par convention référencé à l'aide d'un qualificateur comme `SOAP-ENC` ou `soapenc`. Dans nos exemples, nous utiliserons le qualificateur d'espace de noms `SOAP-ENC` pour nous y référer.

Il est important de comprendre que dans SOAP, les schémas sont utilisés comme des références aux définitions d'éléments de données. Ils ne sont pas utilisés pour valider les données des messages dans le processus SOAP standard, bien que rien ne vous empêche de le faire. Les références aux schémas sont souvent utilisées comme espaces de noms, pour qualifier un élément de donnée sérialisé. Il appartient au développeur ou à la

structure sous-jacente de comprendre l'architecture ou le sens des données et de coder de façon appropriée.

La section 5 SOAP inclut tous les types de données prédéfinis de XML Schema Part 2: Datatypes.[1] Ce schéma définit la plupart des types de données que vous utiliserez, que ce soit directement où comme parties de vos propres types. La pratique courante consiste à déclarer un identifiant d'espace de noms nommé `xsd` et à l'associer à l'espace de noms `http://www.w3.org/2001/XMLSchema`. Comme nous l'avons vu au chapitre précédent, cette déclaration est habituellement faite sous forme d'attribut des éléments SOAP `Envelope` ou `Body`.

Un autre identifiant d'espace de noms couramment utilisé pour l'encodage des données est `xsi`, qui est associé à l'espace de noms `http://www.w3.org/2001/XMLSchema/instance`. Il est plus aisé d'expliquer cet identifiant à l'aide d'un exemple. Imaginez que l'on veuille déclarer que le type de données d'un élément XML particulier est un `float` tel que défini par XML Schema 2001. Nous faisons cela en utilisant l'attribut type de `xsi`, qui est un type de donnée instancié. L'attribut `xsi:type` spécifie le type de la valeur de la donnée encodée. Voici un exemple de son utilisation:

```
<xyz xsi:type="xsd:float">3.14159</xyz>
```

Notez que le schéma 2001 n'est pas le seul utilisé. Les schémas 1999 et 2000 le sont également et d'autres émergeront dans l'avenir. Les implémentations SOAP devraient être capables de gérer toutes les possibilités car les messages contiendront une référence au schéma approprié.

Les identifiants d'espaces de noms qui suivent se retrouvent fréquemment dans la littérature SOAP et dans les implémentations. Jetons un œil à leurs déclarations afin que vous les reconnaissiez:

```
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema/instance/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
```

Terminologie

Pour bien comprendre l'encodage SOAP, il est important de maîtriser la terminologie. Ces termes sont décrits dans les spécifications SOAP, mais allons nous y attarder un peu et voir quelques exemples utilisant à la fois Java et XML.

Le terme *valeur* est utilisé pour décrire la donnée réelle encodée dans un élément XML. La donnée est une chaîne de caractères. L'interprétation de cette chaîne peut être tout ce qu'on peut imaginer en programmation Java: un nombre, un nom, une date ou quelque chose de plus élaboré. Les valeurs rencontrées dans un message SOAP sont toujours associées à un type donné. Nous discuterons des types sous peu. Pour l'instant, il nous suffit de dire que le type de donnée d'une valeur particulière n'est jamais indéfini en SOAP. Il doit toujours y avoir un moyen de déterminer le type correct d'une valeur du message.

1. Ces spécifications sont disponibles à l'adresse *http://www.w3.org/TR/xmlschema-2/*.

SOAP distingue les valeurs simples et composées. Une valeur *simple* ne contient aucune partie nommée mais simplement une donnée. Les valeurs associées aux types des langages de programmation tels que les chaînes, les entiers et les flottants seront tous considérés comme des valeurs simples. Voici quelques exemples en Java avec leur correspondance dans SOAP :

```
int a = 10;                    <a xsi:type="xsd:int">10</a>
float x = 3.14159;             <x xsi:type="xsd:float">3.14159</x>
java.lang.String s = "SOAP";   <s xsi:type="xsd:string">SOAP</s>
```

Chacune des variables ci-dessus contient une valeur simple du type correspondant. La donnée n'est pas divisée en plusieurs parties et aucun autre nom ou mécanisme n'est nécessaire pour y accéder.

Une valeur *composée* contient des données multiples ayant chacune une certaine relation avec les autres. On peut accéder aux données individuelles en indiquant une position ordinale dans une suite de valeurs, tout comme dans un tableau classique. On peut également y accéder à l'aide de valeurs de clés d'un tableau, comme pour une table de hachage. Enfin, on peut y accéder à l'aide des noms des parties constituantes, comme pour une `struct` en C. Quel que soit le mécanisme utilisé, il existe toujours un moyen de distinguer une données particulière à l'intérieur d'une valeur composée et ce mécanisme est appelé un *accesseur*. Il est également possible qu'un des constituants d'une valeur composée soit lui-même composé. Voici quelques exemples de valeurs composées en Java et SOAP :

```
int[3] iTableau = {10, 20, 30};

<iTableau xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:int[3]">
   <val>10</val>
   <val>20</val>
   <val>30</val>
</iTableau>

class Exemple
{
   public int iVal = 10;
   public java.lang.String sVal = "Dix";
}
Exemple exemp = new Exemple();

<Exemple>
   <iVal xsi:type="xsd:int">10</iVal>
   <sVal xsi:type="xsd:string">Dix</sVal>
</Exemple>
```

Dans cet exemple, les accesseurs ordinaux sont utilisés pour accéder aux valeurs de la variable tableau Java `iTableau`. Pour accéder aux parties de la variable Java `exemp`, nous utilisons les accesseurs nommés `iVal` et `sVal`. Nous examinerons plus en détail quelques types composés, y compris les tableaux et les structures, dans la section *Types composés*.

En SOAP, comme dans la plupart des langages de programmation, les valeurs sont associées à un type de données approprié. Nous avons déjà évoqué cela dans notre discussion sur les valeurs simples et composées. Chacune de ces valeurs avait un type associé.

Tout comme en Java, nous devons déclarer des instances de types puis leur assigner des valeurs.

Règles de sérialisation

Les éléments XML des messages SOAP sont soit *indépendants* soit *encapsulés*. En raison de la nature hiérarchique de XML, la plupart des éléments sont encapsulés sous forme de sous-éléments d'autres éléments. Les éléments indépendants, de ce fait, ne sont pas sous-éléments d'autres éléments. Ils apparaissent au niveau supérieur d'une sérialisation.

Toutes les valeurs d'un message SOAP sont encodées en tant que contenu d'un élément. Les valeurs des données ne peuvent apparaître seules, en dehors des limites d'un élément. Cela ne signifie pas pour autant que chaque élément XML contient une valeur. Ainsi par exemple, les types de données composés tels que les tableaux ou les structures contiennent des sous-éléments qui contiennent les valeurs réelles. Les éléments définissant ces valeurs composées ne contiennent pas directement les données. Les types composés seront abordés un peu plus loin.

Références

La section 5 de SOAP permet également aux éléments de faire référence aux valeurs contenues dans d'autres éléments. Aucune valeur n'est fournie dans ce cas avec l'élément. Au lieu de cela, un attribut identifie l'élément dans lequel la valeur réelle peut être trouvée. La valeur doit être contenue dans un élément indépendant, situé au niveau supérieur de la sérialisation.

L'élément contenant la valeur doit contenir un attribut nommé `id` de type `ID`. La valeur de l'attribut `id` est le nom que d'autres éléments utiliserons pour référencer la valeur. Voici un exemple d'un tel élément:

```
<Nom id="nom-1">Englander</Nom>
```

L'élément `Nom` est un accesseur en lui-même parfaitement valide. Il a de plus, un identifiant permettant aux autres élément de référencer sa valeur.

Les éléments accédant à une valeur d'un autre élément ne possèdent pas de valeurs eux-mêmes. Ils sont des éléments vides, contenant un `href` qui fait référence à la valeur. L'attribut `href` est de type `uri-reference` définie dans les spécifications XML Schema. La valeur de cet attribut fait référence à l'attribut `id` de l'élément contenant la valeur. Voici un exemple d'un élément accesseur faisant référence à la valeur de l'élément `Nom` utilisé précédemment:

```
<Surnom href="#nom-1"/>
```

Cette capacité à faire référence aux valeurs d'autres éléments est importante pour deux raisons. Premièrement, elle permet de réduire la taille d'un message SOAP. Imaginez que vous envoyiez un message contenant les noms de la famille `Englander`. Le XML pourrait ressembler à quelque chose comme ceci:

```
<membre>
   <Prenom>Robert</Prenom>
   <Nom>Englander</Nom>
```

```
</membre>
<membre>
   <Prenom>Jessica</Prenom>
   <Nom>Englander</Nom>
</membre>
<membre>
   <Prenom>Arnold</Prenom>
   <Nom>Englander</Nom>
</membre>
   . . .
   . . .
   . . .
```

Comme vous pouvez le voir, il y a là un nombre important de données dupliquées. Les références nous permettent de réduire les répétitions car plusieurs accesseurs peuvent référencer le même élément. Lorsque cela survient, la valeur est dite *multiréférencée*. Voyons ce que cela donne avec les références multiples :

```
<Surnom id="Nom-Famille">Englander</Surnom>
<membre>
   <Prenom>Robert</Prenom>
   <Nom href="#Nom-Famille"/>
</membre>
<membre>
   <Prenom>Jessica</Prenom>
   <Nom href="#Nom-Famille"/>
</membre>
<membre>
   <Prenom>Arnold</Prenom>
   <Nom href="#Nom-Famille"/>
</membre>
   . . .
   . . .
   . . .
```

Dans ce cas, chaque membre de la famille possède un élément `Nom` faisant référence à la valeur de l'élément `Surnom`. Dans cet exemple fabriqué pour l'occasion, nous n'avons bien entendu pas économisé beaucoup d'espace, voire pas du tout. Toutefois, si la valeur des données avait été d'une taille considérablement supérieure, cette technique aurait permis d'économiser un espace non négligeable. L'économie est encore plus évidente lorsque la valeur référencée est un élément composé.

L'économie d'espace n'est pas la seule raison d'utiliser les variables multiréférencées. Cette technique est également utile lors de la sérialisation d'un graphe, d'une collection ou d'objets faisant référence de façon récurrente à un autre objet. Il est dans ce cas important de maintenir ces relations lors de la reconstruction des objets au moment de la désérialisation. Examinons un tel exemple en Java. Le code suivant montre une classe appelée `Employes` contenant des propriétés relatives au prénom et au nom de l'employé, à son titre ainsi qu'à son patron (qui est également un employé). Nous aurons donc un morceau de code définissant un patron nommé `Robert` et trois employés nommés `Benoît`, `André` et `Mylène`, travaillant tous pour `Robert`.

```
class Employe {
   protected java.lang.String _prenom;
   protected java.lang.String _nom;
   protected java.lang.String _titre = "Abeille travailleuse";
   protected Employe _patron;
   public Employe(java.lang.String prenom, java.lang.String nom,
                  Employe mgr) {
      _prenom = prenom;
      _nom = nom;
      _patron = mgr;
   }
   public void affecteTitre(java.lang.String titre) {
      _titre = titre;
   }
   public java.lang.String recupTitre() {
      return _titre;
   }
   public java.lang.String recupPrenom () {
      return _prenom;
   }
   public java.lang.String recupNom () {
      return _nom;
   }
   public Employe recupPatron() {
      return _patron;
   }
}
Employe _robert = new Employe("Robert", "Englander", null);
_robert.affecteTitre("Esclavagiste");
Employe _benoit = new Employe("Benoît", "Zafra", _robert);
Employe _andre = new Employe("André", "Dupont", _robert);
Employe _mylene = new Employe("Mylène", "Phares", _robert);
```

La *figure 3-1* montre les relations entre les trois abeilles travailleuses (Benoît, André et Mylène) et leur esclavagiste de patron (Robert). Très clairement, nous aimerions ne pas avoir à répliquer l'objet référencé par la variable `_robert`. Cela ne montrerait pas correctement les relations entre ces objets. Nous voulons que chaque objet « employe » référence un même objet comme étant son patron. Si l'utilité de cette distinction ne vous est pas évidente, imaginez que robert soit promu Esclavagiste en Chef. Nous ne voulons pas avoir à appeler `recupPatron()` pour chaque objet « employe » puis `affecteTitre()`. Nous voulons n'avoir à modifier que l'objet `_robert`.

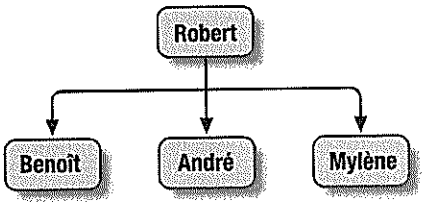


Figure 3-1. Hiérarchie d'employés

Dans SOAP, les variables multiréférencées préservent ce genre de relations entre objets et les graphes de ceux-ci sont donc correctement représentés des deux côtés d'une transaction SOAP. Du XML adapté aux relations de cet exemple pourrait ressembler à ceci :

```
<robert id="Robert">
   <prenom>Robert</prenom>
   <nom>Englander</nom>
   <titre>Esclavagiste</titre>
</robert>
<benoit>
   <prenom>Benoît</prenom>
   <nom>Zafra</nom>
   <titre>Abeille travailleuse</titre>
   <patron href="#Robert"/>
</benoit>
<andre>
   <prenom>André</prenom>
   <nom>Dupont</nom>
   <titre>Abeille travailleuse</titre>
   <patron href="#Robert"/>
</andre>
<mylene>
   <prenom>Mylène</prenom>
   <nom>Phares</nom>
   <titre>Abeille travailleuse</titre>
   <patron href="#Robert"/>
</mylene>
```

Indication du type

Chaque élément contenant une valeur doit également indiqué le type de la donnée. Il existe plusieurs façons d'indiquer le type d'une donnée. Le premier mécanisme consiste à inclure un attribut `xsi:type` comme partie de l'élément. Ceci signifie que l'attribut est nommé type comme défini par l'espace de noms indiqué par `xsi`. La valeur assignée à `xsi:type` doit être un identifiant de type valide tel que `xsd:float`, `xsd:string` etc. Dans le second mécanisme, la valeur peut être un élément d'un tableau ayant déjà défini le type d'un de ses constituants. Dans ce cas, aucune déclaration explicite de type n'est nécessaire pour les valeurs individuelles. Nous verrons cela plus tard lorsque nous évoquerons les tableaux. Enfin, le nom de l'élément lui-même peut être lié un type déterminable par l'examen du schéma XML associé. Cet extrait d'un schéma XML définit un type de données composé :

```
<element name="Automobile" type="Automobile"/>
<typeComplexe name="Automobile">
      <element name="marque" type="xsd:string"/>
      <element name="modele" type="xsd:string"/>
      <element name="annee" type="xsd:int"/>
</typeComplexe>
```

Le type de donnée `Automobile` contient les éléments nommés `marque` et `modele` du type `string`, ainsi que l'annee du type `int`. Examinons maintenant une instance du type `Automobile` basée sur ce schéma :

```
<chouetteVoiture xsi:type="Automobile" >
      <modele>Corvette</modele>
<marque>Chevrolet</marque>
      <annee>1999</annee>
</chouetteVoiture>
```

Ce fragment XML définit une instance du type `Automobile` : une Chevrolet Corvette de 1999. Le schéma pour le type `Automobile` spécifiant déjà les types des parties constituantes, nous n'avons pas eu besoin de déclarer les types dans l'exemple.

Types simples

SOAP fait un usage des types simples comme défini dans XML Schema Part 2 : Datatypes, dans la section « Built-in datatypes » (types de données prédéfinis). Cette section des spécifications traite des entiers, flottants, chaînes etc, ainsi que d'une variété de types de données dérivés des types élémentaires. Par exemple, `positiveInteger` est un type simple basé sur le type `int`, mais contraint à n'accepter que des valeurs positives.

Le fait qu'un type soit déclaré comme prédéfini ne signifie pas qu'il sera automatiquement supporté par chaque implémentation SOAP. La chose la plus judicieuse à faire est de vérifier dans la documentation du système que vous utilisez. Vous serez probablement tranquille avec des types tels que `xsd:int`, `xsd:string` et `xsd:float`. Je pense que vous vous rendrez compte que plusieurs autres sont implémentés à peu près partout, mais vérifiez néanmoins.

Si vous examinez les spécifications XML Schema, vous découvrirez un nombre considérable de types simples. Le *tableau 3-1* montre quelques-uns d'entre eux ainsi que des exemples de valeurs.

Tableau 3-1. Types simples définis par XML Schema

Type	Exemple
int	-41
float	3.14159
string	« Java et SOAP »
positiveInteger	100

Voici les exemples du *tableau 3-1* tels qu'ils peuvent apparaître dans un document XML :

```
<iVal xsi:type="xsd:int">-41</iVal>
<fVal xsi:type="xsd:float">3.14159</fVal>
<sVal xsi:type="xsd:string">Java et SOAP</sVal>
<pVal xsi:type="xsd:positiveInteger">100</pVal>
```

Les types de données sont déclarés explicitement à l'aide de l'attribut `xsi:type` et apparaissent donc clairement. Souvenez-vous cependant que ce n'est pas la seule façon d'associer un élément à un type.

Chaînes

Vous pouvez rencontrer deux types différents de déclarations pour les éléments chaînes. Le premier est `xsd:string`, que nous avons utilisé dans notre exemple précédent. Le second est `SOAP-ENC:string`, qui est un type basé sur `xsd:string` permettant l'usage des attributs `id` et `href` employés pour les valeurs multiréférencées. Ces types chaîne ne sont pas exactement identiques à ceux des langages de programmation, comme `java.lang.String` de Java. Quelques restrictions sont apportées aux types chaîne SOAP, empêchant l'usage de caractères spéciaux comme ceux utilisés pour la formation de balises XML. Il est clair que des caractères comme < et > interfèreraient avec l'ensemble de la structure du document XML s'ils apparaissaient à l'intérieur d'une chaîne. Voyez l'exemple suivant :

```
<value xsi:type="SOAP-ENC:string">incorporer un < n'est pas valide</value>
```

L'inclusion du caractère < dans la valeur de la chaîne pose un problème car il possède un sens spécial en XML. Il va sans dire qu'il est impossible d'encoder une chaîne comme celle-la. Vous devez utiliser des caractères de substitution ou un type autre que `SOAP-ENC:string`. Les séquences d'échappement prévues pour ces caractères, telles que `<` et `>` sont une solution. Une autre façon de résoudre ce problème est d'utiliser l'encodage MIME base64, car l'alphabet base64 ne contient aucun caractère prohibé. Nous examinerons l'encodage base64 sous peu.

Énumérations

L'énumération de valeurs consiste essentiellement à grouper des noms représentant d'autres valeurs. En langage C, le mot-clé `enum` permet d'utiliser des descripteurs en lieu et place de valeurs entières. Ainsi, au lieu d'utiliser les valeurs 1, 2 et 3 pour représenter les couleurs rouge, jaune et vert, vous pourriez définir un type énuméré utilisant les noms des couleurs à la place des valeurs. A l'exécution, le système remplacerait les noms par les valeurs associées. Cela n'est rien d'autre qu'un confort de programmation permettant de produire un code source plus lisible et compréhensible.

Les types énumérés posent néanmoins un problème. En effet, les valeurs utilisées pour les représenter de façon interne ne sont pas standardisées. Il n'existe pas de valeur entière standard pour représenter le mot vert, par exemple. Il y a donc un problème inhérent lors de la transmission de valeurs énumérées d'un système vers un autre. Les valeurs réelles utilisées de façon interne sont plus que probablement très liées à l'application. En langage C, les valeurs sont souvent par défaut une séquence d'entiers démarrant par 0. Dans ce cas, l'ordre dans lequel les noms sont définis dans l'énumération détermine les valeurs. La première entrée est définie comme étant 0, la seconde 1 et ainsi de suite. La syntaxe permet toutefois au programmeur d'ignorer ce comportement et de définir chaque entrée indépendemment. Les valeurs réelles deviennent alors une particularité de programmation ayant peu de chance d'être identique d'un programme à l'autre. Certains langages de programmation comme Java ne supportent même pas les énumérations.[2] Cela n'enlève rien cependant à l'utilité des types énumérés.

---

2. Java, possède naturellement une interface `Enumeration`, qui est un mécanisme pour accéder aux éléments d'une collection. Cette fonctionnalité n'est pas apparentée au type énuméré du C.

SOAP encode les types énumérés en encodant le nom de la valeur. Dans le cas de feux tricolores, l'encodage SOAP ignorera la valeur interne au profit du nom de la couleur. Voici un petit morceau de XML montrant l'encodage de l'état d'un feu tricolore imaginaire :

```
<feuTricolore>
   <etatSignal xsi:type="xsd:string">Vert</etatSignal>
</feuTricolore>
```

L'application émettrice n'encode pas la représentation interne de la valeur `Vert`. Mais elle utilise le mot `Vert` lui-même. C'est au destinataire de résoudre la valeur `Vert` en lui affectant sa propre valeur interne. Il est également possible de définir un type énumération dans un schéma. Voici ce que la définition d'un schéma pour le feu tricolore donnerait :

```
<element name="EtatFeuTricolore" type="EtatFeuTricolore"/>
<typeSimple name="EtatFeuTricolore" base="xsd:string">
   <enumeration value="Rouge"/>
   <enumeration value="Orange"/>
   <enumeration value="Vert"/>
</typeSimple>
```

À l'aide de cette définition de schéma, nous pourrions encoder la valeur `feuTricolore` ainsi :

```
<feuTricolore>
   <EtatFeuTricolore>Vert</EtatFeuTricolore>
</feuTricolore>
```

Encodage binaire

SOAP recommande que les données binaires, référencées dans les spécifications *comme des tableaux d'octets*, soient encodées à l'aide de l'algorithme MIME base64. Comme il n'existe pas de donnée qui ne puisse être représentée sous forme binaire, cela signifie qu'un document SOAP peut encoder n'importe quel type de donnée à l'aide du jeu de caractères base64. Ceci apporte un bénéfice supplémentaire qui est de pouvoir opérer la substitution des caractères ayant un sens particulier sous certains protocoles.

Il ne serait pas approprié de couvrir ici tous les détails de l'encodage base64 mais il peut néanmoins être utile de voir un exemple simple. L'encodage base64 utilise un alphabet composé de valeurs à 6 bits. Chacune de ces valeurs possède un caractère de substitution défini. Le *tableau 3-2* montre l'alphabet base64.

Tableau 3-2. Jeu de caractères base64

valeur à 6 bits	Caractère base64
0-25	A - Z
26-51	a - z
52-61	0 - 9
62	+
63	/
PAD	=

Pour encoder vos données binaires, vous devez les diviser en fragments de 6 bits. Si le nombre total de bits de vos données n'est pas divisible par 6, complétez-les à droite par des zéros. L'encodage base64 a été conçu pour le courrier électronique, qui contraint chaque ligne de données à ne pas dépasser 76 caractères. Dans le monde SOAP, nous ne sommes pas contraints par ces limites et vous n'aurez donc pas à diviser vos données de cette façon. Vous pourrez insérer de nouvelles lignes partout où cela sera nécessaire.

Supposons que nous voulions encoder la chaîne « <data> ». Les deux caractères < et > posent un problème dans l'encodage sous forme de chaîne mais pas en base64. Voyons d'abord la représentation hexadécimale de chaque octet :

Caractère	<	d	a	t	a	>
Valeur hexa	0x3C	0x64	0x61	0x74	0x61	0x3E

Nous allons ensuite la convertir en format binaire et la diviser en fragments de 6 bits. Cela nous donnera exactement 8 fragments et nous n'aurons donc pas à compléter les données avec des zéros. Voici les fragments binaires à 6 bits ainsi que leur équivalents décimaux et les caractères base64 associés :

Valeur 6 bits	001111	000110	010001	100001	011101	000110	000100	111110
Valeur décimale	15	6	17	33	29	6	4	62
Valeur base64	O	F	P	h	d	F	D	+

Nous pouvons maintenant encoder les données en base64 et les représenter en XML :

```
<valeur xsi:type="SOAP-ENC:base64">OFPhdFD+</valeur>
```

Pour décoder, nous convertissons d'abord chaque caractère de la chaîne dans son équivalent à 6 bits. Puis nous regroupons les bits par 8 pour former un octet de donnée décodée. Nous retrouvons en principe notre chaîne d'origine : « <data> ».

L'encodage en base64 peut être très utile pour la transmission de n'importe quel sorte de données binaires. Si vous pensez l'utiliser pour transmettre un fichier bitmap de 10 Mo, il va falloir y réfléchir à deux fois. base64 accroît la taille des données d'à peu près 1/3 ce qui peut être significatif dans certains environnements. Cela ne veut pas dire que cela ne marchera pas et après tout cela pourrait convenir à votre application.

Une technique commune pour traiter les objets de grande taille consiste à envoyer une URL référençant les données plutôt que les données elles-mêmes. Le destinataire peut alors utiliser l'URL pour récupérer les données au moment le plus opportun. C'est une technique adaptée lorsque l'on veut conserver au message SOAP une petite taille, particulièrement si le destinataire peut se passer des données.

> Une autre technique, dont nous discuterons au *chapitre 8*, consiste à envoyer les données sous forme d'attachement MIME. Ceci ne réduit pas la taille globale du message mais maintient les données en dehors de la charge utile SOAP, améliorant ainsi le traitement SOAP.

Accesseurs polymorphes

Le terme *polymorphisme* est fréquemment utilisé en conception et programmation orientés objet. Il signifie qu'une chose (habituellement un objet) peut prendre l'apparence d'une autre. Bien que ce livre ne traite pas de programmation orientée objet, voyons ce qu'est le polymorphisme en Java et examinons ensuite sa signification en SOAP.

Considérons notre classe Java `Employe`. Nous avons utilisé la classe `Employe` pour les patrons comme pour les employés. Dans le monde réel, naturellement, il existe des caractéristiques et des comportements pour les patrons, que l'on ne retrouve pas chez les simples employés (non, je ne veux pas parler de traits de personnalités bien que nous connaissions tous ces différences !). Par exemple, une classe patron peut inclure un ensemble d'employés, objets de son attention (ou de son contrôle, cela dépend de votre point de vue). Nous pouvons créer une classe `Patron` dérivée de la classe `Employe`, puis ajouter les caractéristiques du patron à cette nouvelle classe. Ceci fonctionne car les patrons sont aussi des employés. En d'autres termes, les patrons sont un type d'employés et donc une instance d'un objet `Patron` peut aussi prendre la forme d'un objet `Employe`. Si l'instance du `Patron` est passée à une fonction sous forme de référence à un `Employe`, la méthode réceptrice ne saura pas qu'il s'agit en réalité d'une instance de `Patron`. Elle la verra comme une instance d'`Employe`. Ceci est un exemple de polymorphisme.

Un autre exemple de polymorphisme est le type variant en programmation Microsoft COM (*Component Object Model*). Un variant représente une valeur associée à un indicateur de type de donnée. L'indicateur dit à l'utilisateur quel accesseur est à utiliser pour récupérer les données.

En SOAP, les mécanismes du polymorphisme ne sont pas aussi élaborés qu'ils le sont en Java. Le polymorphisme SOAP ressemble plus au variant COM. Si vous avez examiné les spécifications SOAP, vous aurez peut-être noté que la section sur le polymorphisme ne fait que quelques lignes. Le polymorphisme en SOAP signifie seulement qu'un accesseur comporte une déclaration de type. C'est tout. Le fragment suivant de XML montre un accesseur nommé `quantité`. Dans le premier cas, on suppose que la `quantité` est de type `xsd:int` suivant un schéma quelconque. De ce fait, aucune déclaration de type n'est utilisée. La valeur ne peut être d'un autre type. Dans le second cas, l'accesseur `quantité` inclut une déclaration lui permettant de prendre le type `xsd:float`. La seconde forme est un accesseur polymorphe.

```
<quantité>37</quantité>
<quantité xsi:type="xsd:float">37</quantité>
```

Nous verrons quelques utilisations importantes des accesseurs polymorphes lorsque nous aborderons les tableaux hétérogènes et les types composés génériques.

Types composés

Les spécifications SOAP traitent principalement de deux sortes de types de données composés : les structures et les tableaux. Voici comment les spécifications définissent ces types :

Structure

Valeur composée dont les noms des accesseurs sont les seules distinctions entre les valeurs des membres et sont tous différents.

Tableau

Valeur composée dans laquelle la position ordinale est la seule distinction entre les valeurs des membres.

Nous avons précédemment effleuré ces définitions lorsque nous avons parlé de la terminologie de l'encodage SOAP. Ces structures de données sont communes à de nombreux langages de programmation et il n'y a rien de différent dans la façon dont SOAP les définit.

Structure

De nombreux langages de programmation supportent les structures, bien que de façons parfois différentes. En langage C par exemple, on définit une structure à l'aide du mot-clé `struct`. Ce mot-clé permet essentiellement de définir un type ou une instance de donnée comprenant des parties nommées, dans lequel chaque partie peut être une instance de tout autre type de donnée valide. En C++ on utilise le mot-clé `struct` de la même manière qu'en C. On utilisera toutefois plus souvent une structure spéciale (et plus fonctionnelle) appelée une classe. Il n'existe pas en Java de concept explicite appelé structure, mais comme en C++ on peut utiliser une classe pour fournir quelque chose d'équivalent.

Voici une classe Java contenant des champs représentant les caractéristiques d'une guitare :

```
class Guitare {
   public java.lang.String _fabriquant;
   public java.lang.String _modele;
   public int _annee;
}
```

Ce n'est peut-être pas la meilleure façon de représenter une guitare dans une vraie application Java, mais c'est une façon parfaitement valide et cela ressemble à la `struct` disponible en C et C++. Un fragment de schéma XML correspondant à cette classe pourrait ressembler à cela.

```
<element name="Guitare" type="Guitare"/>
<typeComplexe name="Guitare">
   <element name="fabriquant" type="xsd:string"/>
   <element name="modele" type="xsd:string"/>
   <element name="annee" type="xsd:int"/>
</typeComplexe>
```

Voici un exemple d'une instance sérialisée de la classe `Guitare` utilisant ce schéma :

```
<Guitare>
   <fabriquant>Epiphone</fabriquant>
   <modele>Sheraton II</modele>
   <annee>1997</annee>
</Guitare>
```

Tableaux

Les instances de tableaux SOAP sont déclarées à l'aide du type de donnée `SOAP-ENC:Array`. Les éléments d'un tableau peuvent être de n'importe quel type SOAP valide y compris des tableaux. Les tableaux doivent avoir un attribut `SOAP-ENC:arrayType` spécifiant le type de donnée des éléments ainsi que les dimensions du tableau.

Commençons par la déclaration d'un tableau à une dimension, comportant cinq valeurs à virgule flottante. Les valeurs réelles sont omises pour l'instant (nous y reviendrons plus tard).

```
<quelquesNombres xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:float[5]">
   . . .
   . . .
</quelquesNombres>
```

Un tableau est sérialisé sous forme de séries de valeurs d'éléments, dans lesquelles l'ordre des éléments représente leurs positions ordinales. Le nom réel de l'élément de chaque valeur n'est pas significatif, les membres du tableau étant accédés par leur position. XML exige qu'un nom soit utilisé pour chaque élément du tableau mais SOAP ne se préoccupe pas de ce que ce nom peut être. Vous verrez souvent que chaque élément d'un tableau sérialisé utilise le même nom, mais ceci n'est pas requis. Les spécifications SOAP suggèrent d'utiliser le type de l'élément comme nom. Mais là encore, il ne s'agit que d'une suggestion. Je préfère pour ma part utiliser si possible des noms reflétant le contenu du tableau. Voici un tableau avec les valeurs des données :

```
<quelquesNombres xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:float[5]">
   <nombre>1,234</nombre>
   <nombre>2,345</nombre>
   <nombre>3,456</nombre>
   <nombre>4,567</nombre>
   <nombre>5,678</nombre>
</quelquesNombres>
```

Dans la mesure où les éléments sont sérialisés dans leur ordre d'apparition dans le tableau, vous pouvez déterminer leur position ordinale en les examinant. Comme Java et d'autres langages, SOAP fait débuter les tableaux à l'indice 0. La valeur 4,567 est donc à la position ordinale 3 (le 4ème élément du tableau).

Dans le cas de tableaux multidimensionnels, les éléments sont listés dans l'ordre des lignes. Cela signifie qu'une position ordinale plus éloignée de la droite change plus vite. La *figure 3-2* est un tableau à deux dimensions contenant les valeurs entières 1 à 6. Les dimensions du tableau sont 3 par 2 et la figure montre que la dimension verticale est la majeure et l'horizontale la mineure.

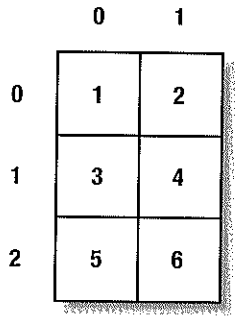


Figure 3-2. Tableau à deux dimensions

L'attribut `SOAP-ENC:arrayType` permet de spécifier la taille de chaque dimension d'un tableau multidimensionnel en les séparant par une virgule. Voici à quoi ressemblerait l'encodage du tableau de la *figure 3-2* :

```
<valeurs xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:int[3,2]">
   <valeur>1</valeur>
   <valeur>2</valeur>
   <valeur>3</valeur>
   <valeur>4</valeur>
   <valeur>5</valeur>
   <valeur>6</valeur>
</valeurs>
```

Les tableaux que nous avons vu jusqu'à présent étaient homogènes : tous les éléments du tableau étaient du même type. Mais il est également possible d'encoder des tableaux hétérogènes, dans lesquels les valeurs sont de types différents. C'est un domaine dans lequel les accesseurs polymorphes montrent leur importance. Pour créer un tableau hétérogène, on assigne à l'attribut `SOAP-ENC:arrayType` la valeur `xsd:ur-type`. Ce type signifie que les types des valeurs du tableau ne sont pas contraints par l'attribut `SOAP-ENC:arrayType` mais que chaque valeur est encodée de façon à inclure son propre type. Voici un tableau contenant des valeurs de types différents :

```
<morceauxMelanges xsi:type="SOAP-ENC:Array"
                  SOAP-ENC:arrayType="xsd:ur-type[3]">
   <morceau xsi:type="xsd:float">3.14159</morceau>
   <morceau xsi:type="xsd:string">SOAP</morceau>
   <morceau xsi:type="xsd:int">8141992</morceau>
</morceauxMelanges>
```

SOAP permet également d'omettre la taille d'une dimension d'un tableau. Cela signifie que le nombre de valeurs d'une dimension particulière d'un tableau peut être déterminé en examinant les membres encodés dans le tableau. Ainsi, la première ligne de l'exemple `morceauxMelanges` peut aussi être écrite comme suit :

```
<morceauxMelanges xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:ur-type[]">
```

En examinant ce XML, nous sommes en mesure de déterminer qu'il s'agit d'un tableau à une dimension et de 3 éléments.

Tableaux partiels

Il peut arriver qu'un tableau ne soit pas complètement rempli. En d'autres termes, il se peut qu'un ou plusieurs éléments n'aient pas de valeurs. Dans le code Java qui suit, le tableau `concurrents` est un tableau de chaînes unidimensionnel de taille 10 mais seuls les éléments en indice 6, 7 et 8 possèdent une valeur :

```
java.lang.String concurrents[] = new java.lang.String[10];
concurrents[6] = "Robert";
concurrents[7] = "Arnold";
concurrents[8] = "Serge";
```

Si nous utilisions l'encodage standard pour ce tableau, il nous faudrait encoder chaque élément, même si la plupart des valeurs sont nulles. Les valeurs nulles ne doivent pas être confondues avec les valeurs vides. Dans le cas de notre tableau de chaînes, il existe une différence entre un élément contenant une chaîne vide et un élément nul. Nous voudrons probablement encoder des chaînes vides car cela montre clairement que les données existent. Dans l'exemple ci-dessus, les valeurs du tableau `concurrents` situées aux positions autres que 6, 7 et 8 sont nulles. Elles ne contiennent aucune instance de `java.lang.String`. Nous pourrions encoder ces éléments nuls en fournissant un attribut `xsi:null` possédant la valeur 1 à chacun d'entre eux. Ces éléments ne contiendront aucune donnée :

```
<concurrents xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10]" >
   <nom xsi:null="1"/>
   <nom xsi:null="1"/>
   <nom xsi:null="1"/>
   <nom xsi:null="1"/>
   <nom xsi:null="1"/>
   <nom xsi:null="1"/>
   <nom>Robert</nom>
   <nom>Arnold</nom>
   <nom>Serge</nom>
   <nom xsi:null="1"/>
</concurrents>
```

Ceci est parfaitement acceptable mais n'est pas très efficace. SOAP permet d'encoder ce type de tableau en omettant tous les éléments ne contenant pas de valeurs. Pour cela, on inclut l'attribut `SOAP-ENC:offset` dans la déclaration du tableau. La valeur de cet attribut représente la position ordinale (à partir de zéro) du premier élément réellement inclus dans le tableau. Nous pouvons donc encoder le tableau `concurrents` comme ceci :

```
<concurrents xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10]"
             SOAP-ENC:offset="[6]">
   <nom>Robert</nom>
   <nom>Arnold</nom>
   <nom>Serge</nom>
</concurrents>
```

Ceci est bien plus efficace que de transmettre toutes les valeurs nulles. Notez que l'attribut `SOAP-ENC:arrayType` continue à spécifier que le tableau possède une taille de 10. Ceci est important car nous voulons que le destinataire de ce tableau utilise la bonne taille.

Tableaux épars

L'utilisation de l'attribut `SOAP-ENC:offset` fonctionne parfaitement si les valeurs présentes dans le tableau partiel sont contiguës, comme elles le sont dans l'exemple des `concurrents`. Mais ce n'est certainement pas le seul scénario possible de tableaux partiellement remplis. Que se passe-t-il si les données sont situées à des positions non contiguës, comme dans le code suivant ?

```
java.lang.String concurrents[] = new java.lang.String[10];
concurrents[1] = "Robert";
concurrents[5] = "Arnol d";
concurrents[8] = "Serge";
```

C'est ce que l'on appelle un tableau épars. SOAP pourvoie aux tableaux épars en autorisant chaque élément transmis à spécifier sa propre position ordinale en incluant un attribut `SOAP-ENC:position`. Nous pouvons donc maintenant encoder le tableau comme ceci :

```
<concurrents xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10]">
   <nom SOAP-ENC:position="[1]">Robert</nom>
   <nom SOAP-ENC:position="[5]">Arnold</nom>
   <nom SOAP-ENC:position="[8]">Serge</nom>
</concurrents>
```

L'encodage des tableaux partiels et épars est potentiellement plus efficace que l'encodage des valeurs nulles dans un tableau partiellement rempli. Vous devrez avoir conscience toutefois, que beaucoup des implémentations SOAP actuelles ne supportent pas ces constructions. Je suis certain qu'un jour, et je l'espère au moment où vous lirez ce livre, les implémentations supporteront les tableaux partiels et épars.

Types composés génériques

Il arrive que le format exact de valeurs composées sérialisées ne soit pas connu à l'avance. Cela peut être parce qu'il n'existe pas de définition de schéma pour le type de données ou en résultat d'une requête générale sur une base de données. Supposons par exemple que vous utilisez SOAP pour interroger une base de données par une requête SQL comme celle-ci :

```
select * from guitares;
```

Au cas où vous ne seriez pas familiers de la syntaxe SQL, cette instruction demande toutes les lignes de la table nommée `guitares` et renvoie toutes les colonnes de toutes les lignes de la table. Nous pourrions toutefois ne pas connaître à l'avance les noms des colonnes ni même savoir combien la table en comporte. Le *tableau 3-3* représente une base de données très simple nommée `guitares`, contenant deux lignes. La table contient trois colonnes : `fabriquant`, `modele` et `annee`.

Tableau 3-3. La table guitares

Fabriquant	Modèle	Année
Gibson	Les Paul	1958
Fender	American Stratocaster	2001

Même si nous ne savons pas à l'avance quelles colonnes nous seront retournées par notre requête SQL, une réponse SOAP valide pourra contenir les données sérialisées suivantes :

```
<guitares>
   <guitare>
      <fabriquant xsi:type="xsd:string>Gibson</fabriquant>
      <modele xsi:type="xsd:string">Les Paul</modele>
      <annee xsi:type="xsd:int">1958</annee>
   </guitare>
   <guitare>
      <fabriquant xsi:type="xsd:string>Fender</fabriquant>
      <modele xsi:type="xsd:string">American Stratocaster</modele>
      <annee xsi:type="xsd:int">2001</annee>
   </guitare>
</guitares>
```

Ce code XML contient des données composées génériques. C'est au destinataire de comprendre comment les traiter. Les structures et les tableaux peuvent être tous deux utilisés dans les types génériques. Le format de chaque élément `guitare` ressemble à une structure SOAP et l'élément `guitares` à tout l'air d'un tableau, même s'il n'a pas été déclaré avec `xsi:type="SOAP-ENC:Array"`. Les éléments de chaque guitare utilisent des accesseurs polymorphes, dans la mesure où leurs types de données peut ne pas être connu d'avance.

Valeurs par défaut

Les spécifications SOAP traitent brièvement de la prise en charge des valeurs par défaut. Il est dit que l'omission d'un élément d'une donnée transmise, peut impliquer l'usage d'une valeur par défaut. Le langage C++ fournit une bonne analogie sur ce sujet, dans la mesure où il permet de définir des valeurs par défaut pour les paramètres de méthodes, autorisant ainsi l'appelant à omettre les valeurs de ces paramètres. Voici à quoi ressemble une telle définition de méthode en C++ :

```
int recupValeur (int param1, int param2 = 1);
```

Içi, la valeur par défaut pour `param2` est 1. Si l'appelant ne fournit pas de valeur pour ce paramètre lors de l'appel de `recupValeur()`, le système utilise la valeur 1. Bien que cette syntaxe n'existe pas en Java, il est quand même possible de définir des transactions SOAP supportant les valeurs par défaut.

Supposons qu'un système de prise de commandes attende un élément composé nommé `commande`, contenant une quantité, un numéro d'article et une description. Nous voudrions que la valeur par défaut pour la quantité soit 1 et que l'expéditeur ne soit donc pas obligé d'indiquer une quantité s'il ne commande qu'un seul article. Pour cela, nous sérialiserons une commande ainsi :

```
<commande>
   <article>
      <quantite>2</quantite>
```

```
        <articleNum>1020304</articleNum>
        <description>Truc sympa</description>
    </article>
    <article>
        <articleNum>1020305</articleNum>
        <description>Autre truc</description>
    </article>
</commande>
```

Nous n'avons pas inclus d'élément quantité pour le second article. Au lieu de cela, nous attendons du destinataire qu'il utilise la valeur par défaut. Le point important est que SOAP ne définit pas de valeurs par défaut ni n'en exige. Il appartient à l'expéditeur et au destinataire de faire le nécessaire.

L'attribut SOAP Root

Les listes chaînées, les arbres et les graphes orientés sont des structures de données classiques. Nous avons toutefois encore à voir le mécanisme de sérialisation identifiant le point de départ ou racine de telles structures. L'attribut SOAP root est destiné à cet usage. Assigner la valeur 1 à l'attribut `SOAP-ENC:root` l'identifie comme la racine d'une structure telle qu'une liste chaînée.

Voici un extrait de code Java montrant les nœuds d'une liste chaînée simple d'entiers, suivie d'une instance de cette liste contenant trois nœuds. Les nœuds contiennent les valeurs 50, 60 et 70 et sont placés en ordre croissant dans la liste, le nœud contenant la valeur 50 constituant la tête (ou racine) de la liste.

```
class Nœud {
    int _valeur;
    Nœud _suivant;
    public Nœud(int valeur, Nœud suivant) {
        _valeur = valeur;
        _suivant = suivant;
    }
}
Nœud n3 = new Nœud(70, null);
Nœud n2 = new Nœud(60, n3);
Nœud n1 = new Nœud(50, n2);
```

La variable `n1` est la racine de la liste. Voici à quoi ressemblerait la sérialisation de cette liste :

```
<liste>
      <nœud>
         <valeur id="nœud-3"> <xsi:type="xsd:int">70</valeur>
         <suivant xsi:null="1"/>
      </nœud>
      <nœud>
         <valeur id="nœud-2"> <xsi:type="xsd:int">60</valeur>
         <suivant href="#nœud-3"/>
```

```
        </nœud>
        <nœud>
            <valeur SOAP-ENC:root="1" id="nœud-1"> <xsi:type="xsd:int">50</valeur>
            <suivant href="#nœud-2"/>
        </nœud>
    </liste>
```

Nous avons utilisé les références des valeurs pour chaîner chaque nœud avec le suivant et l'attribut `SOAP-ENC:root` pour indiquer la tête de liste.

4

Services de type RPC

Bien que SOAP ne soit pas limité à un style particulier d'informatique distribuée, il se prête bien au modèle RPC. D'ailleurs, c'est tout ce que l'on en attend. D'après les spécifications SOAP, l'un des objectifs de sa conception est d'encapsuler et d'échanger des appels RPC. Cette approche s'accorde bien à la programmation Java, les appels de méthodes d'objets Java pouvant être aisément traduits en appels RPC. Nous débuterons ce chapitre en examinant la structure d'une requête RPC SOAP et des messages de réponse. À partir de là, nous implémenterons quelques services RPC en Java et les déploierons à la fois sous Apache SOAP et GLUE. Bien sûr, nous écrirons le code Java pour utiliser ces services.

Éléments RPC SOAP

La création d'une requête RPC SOAP utilise la structure SOAP et l'encodage décrits aux chapitres 2 et 3. Vous n'avez pas besoin de XML ou de styles d'encodage de données supplémentaires pour effectuer des RPC. Voyons ce qui est nécessaire pour représenter un appel de méthode RPC en SOAP :

- Objet cible
- Nom de méthode
- Paramètres de méthode
- Données d'en-tête SOAP

URI de l'objet cible

L'URI de l'objet cible est essentiellement l'adresse de la ressource que nous souhaitons utiliser dans le service. Vous voyez des adresses de ressources à chaque fois que vous naviguez sur le Web : vous affichez une page en spécifiant l'adresse de la page qui est une ressource. Pour afficher par exemple, la page décrivant l'auteur du livre O'Reilly sur les JavaBeans, vous utiliserez l'URL *http://www.oreilly.fr/catalogue/ejb3.auteur.html*. Le protocole, l'adresse du serveur et la ressource sont regroupés. Découpons-les pour en examiner les composants. L'adresse du serveur est *www.oreilly.fr*, la page de ressource HTML est */catalogue/ejb3.auteur.html* et le protocole est *http*.

C'est exactement ce que nous devons faire pour spécifier l'objet cible d'une requête RPC SOAP, avec une différence : au lieu d'une ressource de type page HTML, on utilise le nom de l'objet cible. Le nom de l'objet cible est acheminé par la couche de transport (par un en-tête HTTP par exemple) et non par le message SOAP lui-même. C'est la seule partie d'un message RPC SOAP acheminée hors du message, à l'exception de l'attribut `SOAPAction`. Dans un serveur Apache SOAP, l'objet cible d'un message RPC est habituellement `/soap/rpcrouter`. Cet URI représente la ressource qui route les messages RPC SOAP dans Apache SOAP. Pour un serveur GLUE, la ressource cible est habituellement `/glue`, suivi du nom du service lui-même. Nous parlerons des en-têtes HTTP pour chacun de ces deux serveurs un peu plus loin.

Nom de méthode

Le nom de méthode est comme vous l'aurez deviné, le nom de la méthode à invoquer. Plus haut dans le chapitre 2, nous avons évoqué un service fournissant la température d'une station météo. Ce service exposait une méthode nommée `DonneTemperatureCourante`. Si vous préférez réfléchir en Java, vous pouvez considérer le nom de méthode comme la méthode publique d'une instance d'un service.

Dans du XML SOAP, la méthode invoquée est packagée comme une struct SOAP[1] contenant un accesseur pour chaque paramètre. Le nom de méthode `struct` est un descendant immédiat de l'élément SOAP `Body`. Examinons de nouveau l'exemple `DonneTemperatureCourante` :

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
        <m:DonneTemperatureCourante xmlns:m="StationMeteo">
              <m:echelle>Celsius</m:echelle>
        </m:DonneTemperatureCourante>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Vous noterez que le nom de la méthode invoquée est qualifié par un espace de noms utilisant le nom du service. L'espace de noms est supposé être un URI fournissant le service cible avec un mécanisme pour interpréter la signification de la méthode. L'URI doit normalement pointer vers une ressource décrivant la méthode, mais cet exemple pourra fonctionner tel quel. Mais de quel objet service sommes nous en train de parler ? Pas de l'objet cible lui-même mais d'un autre objet, connu de l'objet cible sous un nom particulier. Dans ce cas, le nom de l'objet service est `StationMeteo`. Le mécanisme utilisé pour apparier les noms des objets service et leurs implémentations physiques n'est pas indiqué dans les spécifications SOAP. Chaque implémentation procède à sa manière, comme nous le verrons dans la section *Déploiement d'un service*.

---

1. Le terme `struct` est utilisé ici pour signifier la hiérarchie d'éléments XML utilisée pour représenter une méthode de service et ses paramètres. Que les programmeurs C ne confondent pas ceci avec les struct C qui ont bien entendu une signification différente.

La signature de méthode peut également être incorporée comme partie de la structure mais ce n'est pas nécessaire. En réalité, les spécifications SOAP ne décrivent pas le format de la signature de méthode et les signatures ne seront pas utilisées dans ce livre (et ne sont pas utilisées par les serveurs dont je parle). Je les mentionne uniquement afin que vous ne soyez pas surpris si vous rencontrez une implémentation les utilisant.[2]

Paramètres de méthode

Les paramètres de méthodes sont construits comme des accesseurs et sont des éléments immédiatement descendants de la structure de méthode. Ceci est vrai pour tous les paramètres `in` et `in/out`. Les paramètres `out` n'apparaîtront pas dans la structure d'invocation de méthode. SOAP permet le traitement de méthodes sans paramètres, comme nous l'avons vu au chapitre 3. Il appartient à l'application de décider de traiter ou non un appel sans paramètres. Si l'absence de paramètres n'est pas permise, l'application devra répondre par une faute SOAP. Chaque accesseur figure à la suite du nom de paramètre pour la méthode englobante et son type est le même que celui de la méthode de service correspondante.

L'enveloppe SOAP précédente possède un paramètre pour la méthode `DonneTemperatureCourante`. Le paramètre est nommé `echelle` et est utilisé pour spécifier l'échelle de température qui doit être employée pour renvoyer le résultat. Ce paramètre est qualifié par un espace de noms utilisant le nom du service, ce qui revient à utiliser un nom de schéma pour qualifier l'accesseur. Cela signifie que l'élément `echelle` est présenté selon sa définition dans l'espace de noms `StationMeteo`. Nous aurions pu choisir d'inclure un attribut `xsi:type` dans l'`echelle`, mais ne pas le faire est parfaitement valide. Tant que le type correspond au type spécifié dans le schéma référencé par l'espace de noms, il n'est pas nécessaire de le préciser dans le message SOAP. Vous pouvez même envoyer un type différent du moment que vous en spécifiez sa nature.

Examinons un peu de code Java et voyons comment le faire correspondre à un appel de méthode RPC SOAP. La classe `CAC40` représente le cours de l'indice boursier du même nom. `CAC40` contient une méthode nommée `donneNiveauActuel()` qui renvoie le niveau de l'indice pour la session en cours. Elle possède également une méthode nommée `donnePrix()` qui renvoie le prix actuel d'une action. `donnePrix()` reçoit un paramètre de type chaîne nommé `action`, représentant le symbole de l'action et un paramètre de type chaîne `devise`, spécifiant la devise à utiliser lors de l'envoi du prix. Ce dernier est retourné sous forme d'un `flottant`.

```
public class CAC40 {
   private float _niveau;
   public float donneNiveauActuel() {
      return _niveau;
   }
   public float donnePrix(String action, String devise) {
      float resultat;
      // détermine le prix de l'action et le renvoie
      // dans la devise indiquée
```

2. Je doute que vous rencontriez un jour une signature de méthode comme partie d'un message SOAP. Il ne semble pas que ceci soit accepté et ce n'est implémenté dans aucun des moteurs SOAP qu'il m'a été donné de voir.

```
        return resultat;
    }
}
```

Supposons maintenant que nous avons déjà déployé un service nommé CAC implémenté par la classe CAC40. Pour invoquer la méthode donneNiveauActuel() du service CAC, nous utilisons une enveloppe SOAP ressemblant à cela :

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
        <m:donneNiveauActuel xmlns:m="CAC">
        </m:donneNiveauActuel>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Il n'est point besoin de paramètres, la méthode donneNiveauActuel() n'en attendant pas. Appelons maintenant la méthode donnePrix(), en lui fournissant le symbole FTE (France Telecom) et la devise EUR (Euro) comme paramètres. Voici à quoi l'enveloppe ressemblerait :

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
        <m:donnePrix xmlns:m="CAC">
           <m:action>FTE</m:action>
           <m:devise>EUR</m:devise>
        </m:donnePrix>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Réponse de la méthode

La réponse de la méthode prendra également la forme d'une struct SOAP. La structure de la méthode contient un accesseur pour la valeur de retour, suivi d'accesseurs pour chaque valeur de paramètre out et in/out présentés dans leur ordre d'apparition dans la signature de la méthode,[3] si elle existe. Les noms des accesseurs aux valeurs out et in/out correspondent à leurs noms dans la signature de méthode mais les noms des valeurs de retour n'ont pas d'importance. Je préfère pour ma part donner aux valeurs de retour un nom reflétant le nom de la méthode. J'ai tendance à utiliser pour les méthodes de services, un style verbe-nom me permettant de baptiser l'accesseur de la valeur de retour à l'aide du nom. Si mon nom de service est par exemple calculeTaille, je nommerai alors l'accesseur de la valeur de retour taille.

3. Comme mentionné plus haut, la signature de méthode référencée dans les spécifications SOAP ne semble pas avoir été implémentée. Elle est mentionnée ici seulement à titre d'information.

Il est de plus en plus courant de nommer la structure de la valeur de retour en concaténant le mot `Response`[4] et le nom de méthode invoquée. Ainsi, dans le cas d'une structure nommée `calculeTaille`, la structure contenant le résultat s'appellera `calculeTailleReponse`. Bien que ceci soit une convention répandue, ce n'est pas indispensable et vous vous apercevrez que certaines implémentations n'en tiennent pas compte.

Voici l'enveloppe SOAP qui pourrait être renvoyée à l'invocation de la méthode `donneNiveauActuel` :

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
       <m: donneNiveauActuelReponse xmlns:m="CAC">
            <m:niveauActuel>3904.15</m:niveauActuel>
       </m:donneNiveauActuelReponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Réponse de faute

Si un appel de méthode RPC SOAP échoue, une faute SOAP doit être renvoyée. Voyons un exemple. Supposons que nous appelons la méthode `donnePrix` du service `CAC` en spécifiant un paramètre de devise invalide. Le service pourra renvoyer une faute comme celle-là :

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV=
      "http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
     <SOAP-ENV:Fault>
        <faultcode>SOAP-ENV:Client</faultcode>
        <faultstring>Erreur client</faultstring>
        <detail>
           <m:cac40detailsfaute xmlns:m="CAC">
              <message>Devise incorrecte</message>
              <errorcode>1234</errorcode>
           </m:cac40detailsfaute>
        </detail>
     </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Un appel de méthode RPC SOAP peut échouer ou réussir mais pas les deux. On ne verra donc jamais à la fois une réponse de méthode et une réponse de faute.

---

4. NdT : nous utiliserons quant à nous le terme Reponse pour conserver la cohérence de la traduction des noms de classes et de méthodes.

Données d'en-tête optionnelles

Traiter du RPC requiert parfois une information qui ne fait pas partie de la signature de la méthode. Il peut être par exemple nécessaire d'envoyer un nom d'utilisateur à un service. Même s'il n'est normalement pas très approprié de l'envoyer sous forme de paramètre, il n'en sera pas moins requis par l'application réceptrice. Ce type d'information peut être encodé dans l'en-tête SOAP, comme décrit dans la section *L'élément Header* au chapitre 2. Il peut également exister des données liées à la couche de transport. Vous pourriez par exemple vouloir envoyer un cookie dans votre en-tête HTTP pour spécifier la session HTTP à utiliser pour cette invocation de méthode. Dans tous les cas, ces données supplémentaires doivent être encodées hors des limites du corps SOAP.[5]

Activation de service

La durée de vie d'un objet est une notion importante dans tout système distribué et l'on s'y réfère parfois comme à la *portée du service* ou l'*activation du service*. Quelle que soit la terminologie utilisée, elle englobera les différents modèles de durée de vie d'un objet disponibles sur le système. Dans les systèmes SOAP basés sur HTTP, il existe généralement trois modèles d'activation de service :

Activation de service au niveau requête

Lors de chaque requête faite à un objet donné d'un service, une nouvelle instance de l'objet est créée et persiste jusqu'à la fin de la requête. Cette façon d'activer un service est utile pour les objets de service n'ayant pas d'état à maintenir.

Activation de service au niveau application

Une instance unique de l'objet traite chaque invocation de méthode pour toute la durée de vie du service. Cette façon d'activer un service peut être utilisée dans le cas de services ayant besoin de maintenir un état ou qui utilisent beaucoup de ressources lors de la création de l'objet.

Activation de service au niveau session

Une instance unique de l'objet est utilisée pour toutes les invocations de méthodes pendant la durée de vie de la session. La notion de session n'est pas nécessairement standard comme vous le savez peut-être. Il est possible d'implémenter des sessions spécifiques à vos applications en générant et en transmettant des jetons ou l'on peut également tirer parti d'approches existantes comme l'utilisation de cookies. Ce sera certainement l'un des cas pour lesquels vous aurez besoin de passer des informations d'en-tête additionnelles.

Les spécifications SOAP ne définissent aucune exigence ou comportement concernant la durée de vie des objets de service. Il n'existe pas de réelles obligations pour les implémentations SOAP de supporter les trois modèles décrits ci-dessus. Ces modèles d'activation existent cependant également hors de SOAP et vous pouvez vous attendre à les trouver dans beaucoup si ce n'est toutes les implémentations SOAP disponibles. Chaque implémentation peut avoir sa propre façon de mettre en place un modèle d'activa-

5. En réalité, envoyer des données spécifiques à la couche de transport utilisée, comme le sont les cookies, n'est probablement pas une si bonne idée car cela vous lie à cette couche. D'autres données de session indépendantes du transport seraient plus appropriées.

tion, mais les comportements seront probablement les mêmes. Vous vous apercevrez certainement que le modèle d'activation d'un service est spécifié comme élément de son déploiement. Nous verrons ce sujet en pratique lorsque nous parlerons du déploiement de service plus tard dans ce chapitre.

Un service élémentaire

Concevons un service élémentaire que nous appelerons `ServiceCompteurAppels`. Ce service enregistre le nombre d'appels de méthode qu'il reçoit. Il n'est pas vraiment utile en soi, mais il nous donnera l'opportunité d'avoir un service opérationnel et de voir comment les services sont déployés et comment leur activation fonctionne. Les classes de cet exemple se trouvent dans le package `javasoap.livre.ch4`. Assurez-vous de créer sur votre système le répertoire approprié à ce package et de le rendre accessible *via* le classpath utilisé par vos systèmes serveur et client.

Voici le code source de la classe `CompteurMethode`, qui implémente `ServiceCompteurAppels`. Vous noterez que le nom de la classe Java n'est pas nécessairement le même que celui du service.

```
package javasoap.livre.ch4;
public class CompteurMethode {
   compteur
   public CompteurMethode() {
      compteur = 0;
   }
   public int getCompteur()
      return compteur;
   }
   public boolean faitQuelqueChose() {
      compteur ++;
      return true;
   }
}
```

Comme vous pouvez le voir, le code Java ne présente rien de très particulier. `CompteurMethode` contient une variable nommée `compteur` qui enregistre le nombre d'appels de méthode effectués pendant la durée de vie de l'objet. La classe contient une méthode nommée `faitQuelqueChose()` qui incrémente la variable `compteur` et renvoie une valeur booléenne indiquant si l'appel a réussi. On trouve également une méthode nommée `getCompteur()` retournant le nombre de fois où `faitQuelqueChose()` a été appelée. Nous avons baptisé la méthode `getCompteur()`[6] en accord avec les conventions de nommage JavaBeans d'une propriété entière en lecture seule nommée `compteur`.

Nous suivons les conventions JavaBeans pour plusieurs raisons. Premièrement, il est bon de suivre des conventions standard à chaque fois que cela est possible. Plus important que cela, nous verrons plus tard que certaines implémentations SOAP peuvent faire un usage des propriétés JavaBeans sans requérir de travail supplémentaire de la part du programmeur.

---

6. NdT : Le nom de la méthode `getCompteur()` mélange l'anglais et le français, mais ce n'est qu'à ce prix que nous pouvons être en accord avec la convention de nommage des JavaBeans.

Vous noterez que notre classe de service utilise un constructeur sans arguments. Ceci est important. Les implémentations SOAP Java chargent les classes de services dynamiquement, en appelant habituellement `Classe.newInstance()`. Cette méthode requiert de la classe qu'elle ait un constructeur sans arguments. Bien entendu, si nous n'en avions pas explicitement inclus un, un constructeur sans arguments par défaut aurait été automatiquement créé. Nous incluons le constructeur explicitement pour mettre en lumière le fait qu'il est nécessaire, quelle que soit la manière de l'inclure.

Avant d'aller plus loin, examinons l'enveloppe SOAP que nous allons utiliser pour encapsuler l'invocation de la méthode `faitQuelqueChose()` :

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
         <m:faitQuelqueChose xmlns:m="ServiceCompteurAppels">
         </m:faitQuelqueChose>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Le premier élément enfant du Body SOAP est la méthode invoquée. L'élément est qualifié par un espace de noms utilisant le nom du service (et non le nom de la classe implémentant le service). Il n'est point besoin de paramètres pour l'élément `faitQuelqueChose`, puisqu'il n'existe pas de paramètres dans la méthode `faitQuelqueChose()` de notre classe de service.

Déploiement du service

Le déploiement de service n'est pas traité par les spécifications SOAP et chaque implémentation possède donc sa propre procédure. Nous allons examiner le déploiement de service en utilisant deux implémentations : Apache SOAP et GLUE.

Déploiement sous Apache SOAP

Sous Apache SOAP, vous devez créer un *descripteur de déploiement*, qui se présente sous la forme d'un fichier XML contenant les informations sur le service et la classe Java qui l'implémente. Jetons un œil au descripteur de déploiement pour le `ServiceCompteurAppels`, implémenté par la classe Java `javasoap.livre.ch4.CompteurMethode` :

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:ServiceCompteurAppels">

   <isd:provider type="java"
         scope="Application"
         methods="faitQuelqueChose getCompteur">
     <isd:java class="javasoap.livre.ch4.CompteurMethode"
         static="false"/>
   </isd:provider>
   <isd:faultListener>
```

```
    org.apache.soap.server.DOMFaultListener
    </isd:faultListener>
    <isd:mappings>
    </isd:mappings>
</isd:service>
```

L'élément XML le plus extérieur est appelé `service`, qui est qualifié à l'aide de l'identifiant `isd`, représentant l'espace de noms `http://xml.apache.org/xml-soap/deployment`. L'élément `service` possède également un attribut `id` auquel est assigné le nom unique du service déployé. J'ai préfixé le nom du service, `ServiceCompteurAppels` par `urn`. Ce n'est pas indispensable mais c'est une pratique courante que je suivrai, au moins dans cet exemple. Sachez simplement qu'on peut l'omettre.

Le premier élément enfant est nommé `provider` et décrit l'implémentation du service. Comme tous les éléments enfants du `service`, il est qualifié par un espace de noms à l'aide de l'identifiant `isd` de l'élément parent. Trois attributs sont utilisés pour l'élément `provider`. Le premier est `type`, qui indique à Apache SOAP quel type d'implémentation le service utilise. Nous lui assignons la valeur `java`. (Apache SOAP permet d'autres implémentations mais nous ne les traiterons pas.) L'attribut `scope` spécifie le modèle d'activation du service déployé. Les valeurs possibles pour cet attribut sont `Application`, `Session` et `Request`. Elles correspondent aux modèles d'activation décrits précédemment. Nous utilisons pour l'instant l'activation au niveau `Application`. Nous expérimenterons les autres plus tard. Le troisième attribut de l'élément `provider` est `methods`. Cet attribut liste les noms des méthodes exposées par le service. Ces noms de méthodes doivent correspondre à ceux de la classe Java qui implémente le service car SOAP utilise le chargement de classe dynamique et la réflexion Java pour effectuer les appels de méthodes. Les noms de méthodes sont séparés par des espaces. Pour cet exemple, nous exposons les méthodes `faitQuelqueChose` et `getCompteur`.

L'élément `provider` contient un élément fils décrivant un peu plus les détails de l'implémentation du service. Nous avons assigné à l'attribut `type` la valeur `java`, qui correspond au nom de l'élément enfant `java`. L'élément `java` possède deux attributs et aucune donnée. L'attribut `class` se voit assigné le nom complet de la classe Java qui implémente le service, `javasoap.livre.ch4.CompteurMethode` dans ce cas. Un attribut nommé `static` indique si les méthodes de la classe d'implémentation du service sont statiques.

L'élément suivant du descripteur de déploiement est un élément enfant du `service` nommé `faultListener`. La valeur de cet élément est le nom complet de la classe Java utilisée pour créer un objet `faultListener`. Cet objet est responsable de la construction des fautes SOAP à retourner si les conditions d'une faute sont remplies. Nous utilisons la classe `org.apache.soap.server.DOMFaultListener`, fournie par Apache SOAP. Au chapitre 7 nous verrons plus attentivement les fautes SOAP et écrirons notre propre `faultListener`.

Le dernier élément est appelé `mappings`. Au chapitre 6 nous utiliserons cette section pour définir des classes Java prenant en charge la sérialisation et la désérialisation des types Java personnalisés. Nous n'en avons pour l'instant pas besoin et il n'y a donc rien dans la section `mappings`.

Nous devons ensuite utiliser le descripteur de déploiement pour déployer notre service. Le fichier du descripteur de déploiement est nommé *ServiceCompteurAppels.dd*. Son

nom n'a pas d'importance du moment qu'il est reconnaissable et présente un sens. Au point où nous sommes arrivés, vous devez démarrer votre serveur si ce n'est pas déjà fait. Apache SOAP contient une application Java que vous pouvez utiliser en ligne de commande pour déployer un service. Cette application est appelée le *client gestionnaire de service* (`org.apache.soap.server.ServiceManagerClient`). Elle peut déployer, supprimer et lister des services. Voici à quoi ressemble la commande de déploiement d'un service :

```
java org.apache.soap.server.ServiceManagerClient
  http://localhost:8080/soap/servlet/rpcrouter deploy ServiceCompteurAppels.dd
```

Le premier paramètre est l'URL du routeur RPC Apache SOAP. J'utilise un serveur Tomcat Apache qui accepte les connexions sur le port 8080. Le serveur est une machine située sur le réseau local et nommée `localhost`. Le second paramètre, `deploy`, indique au client gestionnaire de service que nous sommes en train de déployer un service. Le dernier paramètre, `ServiceCompteurAppels.dd`, est le nom du fichier contenant le descripteur de déploiement. Si tout se passe bien, vous ne verrez aucune sortie produite par le client. Vous reviendrez simplement à l'invite de commande. Si vous voyez des exceptions ou autres messages d'erreur, examinez-les attentivement. Les problèmes proviennent souvent d'un classpath incorrect ou du fait d'avoir lancé la commande depuis un répertoire autre que celui contenant le fichier du descripteur de déploiement.

Vous pouvez vérifier que le service a été déployé avec succès en utilisant de nouveau le client gestionnaire de service. Nous allons lui demander cette fois de lister tous les services déployés :

```
java org.apache.soap.server.ServiceManagerClient
  http://localhost:8080/soap/servlet/rpcrouter list
```

Le premier paramètre est le même qu'auparavant : l'URL du routeur RPC Apache SOAP. `list` indique au gestionnaire de service de lister les services actuellement déployés. Vous devriez obtenir la réponse suivante :

```
Deployed Services:
    urn:ServiceCompteurAppels
```

Apache SOAP préserve les informations sur les services déployés et donc, si vous arrêtez le serveur Apache SOAP et le redémarrez plus tard, les services seront toujours disponibles. Il ne préserve pas en revanche l'état des objets implémentant ces services. Il démarreront comme pour la première fois, avec une nouvelle durée de vie.

L'outil d'administration d'Apache SOAP

Vous trouverez peut-être plus aisé d'utiliser l'interface web d'administration d'Apache SOAP. L'URL de cette interface contient l'adresse et le port du serveur hébergeant Apache SOAP, suivis de la ressource d'administration SOAP. J'exécute Apache SOAP sur `localhost` sur le port 8080 et l'URL est donc *http://localhost:8080/soap/admin/index.html*. La *figure 4-1* montre la page d'administration principale d'Apache SOAP.

Le bouton List vous emmène sur une page listant tous les services déployés. Vous verrez `urn:ServiceCompteurAppels` listé, comme on le voit *figure 4-2*. Le lien `urn:ServiceCompteurAppels` vous emmène sur une page montrant les détails du déploiement de notre service, comme on le voit *figure 4-3*. Vous pouvez voir tous les détails du déploiement tels que nous les avons spécifiés dans le descripteur de déploiement.

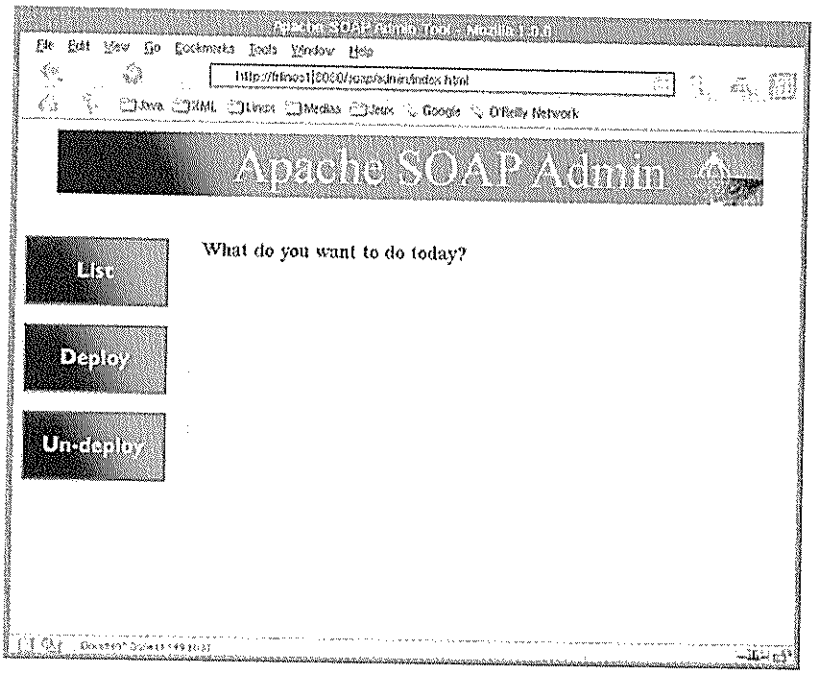


Figure 4-1. Page d'administration d'Apache SOAP

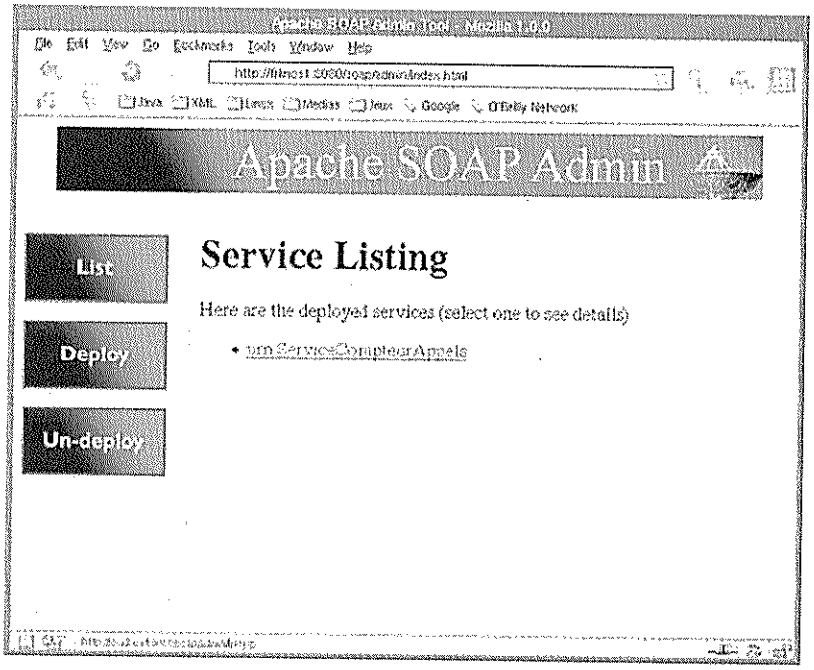


Figure 4-2. Services actuellement déployés

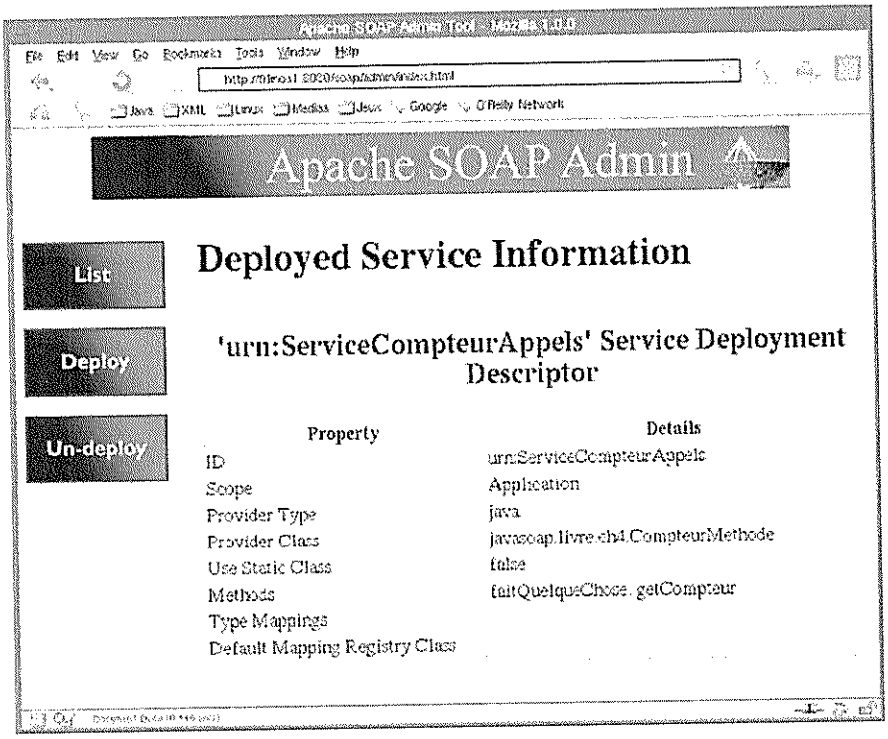


Figure 4-3. Détails des services

Vous pouvez également déployer et supprimer des services *via* cette interface. Si vous cliquez sur Un-deploy, vous verrez une liste des services déployés. Pour supprimer un service, cliquez simplement sur son lien. Si vous le souhaitez, supprimez le service `urn:ServiceCompteurAppels`. Vous pouvez également utiliser le client gestionnaire de service en ligne de commande pour supprimer un service :

```
java org.apache.soap.server.ServiceManagerClient
  http://localhost:8080/soap/servlet/rpcrouter undeploy urn:ServiceCompteurAp-
pels
```

Maintenant que le service n'est plus déployé, cliquez sur le bouton Deploy dans l'interface web. La *figure 4-4* montre la page de déploiement de service. L'information nécessaire est déjà présente.

Pour déployer un service, faites défiler la page vers le bas jusqu'au bouton Deploy. Cliquez dessus et le service sera déployé. Vous pouvez utiliser l'interface web pour examiner les services déployés et vous assurez que ce sont les mêmes qu'auparavant. Apache SOAP ne fait aucune validation des informations qui lui sont fournies et vous devrez donc faire les vérifications vous-même.

Il y a ici un problème évident de sécurité. Voulez-vous autoriser n'importe qui connaissant l'URL appropriée, à déployer et supprimer des services ? Probablement pas. Vous devrez utiliser, dans un environnement de production, les fonctionnalités de sécurité de la technologie hébergeante.

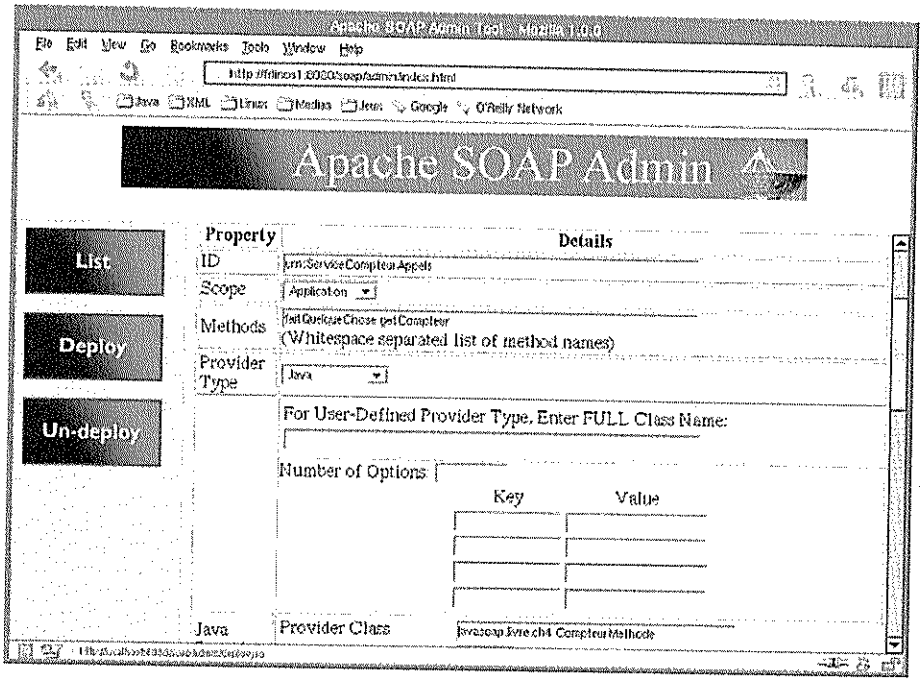


Figure 4-4. Déploiement d'un service

Déploiement sous GLUE

Nous pouvons utiliser la même classe Java pour implémenter `ServiceCompteurAppels` sous GLUE, car comme Apache SOAP, GLUE ne nécessite aucun codage supplémentaire pour les services. Toutefois, le processus de déploiement de services GLUE est assez différent de celui d'Apache SOAP et un peu plus simple. GLUE fournit deux façons de déployer des services. Nous allons le faire en écrivant une application autonome qui démarrera une instance du serveur HTTP intégré à GLUE puis utilisera les classes et méthodes GLUE relatives à la publication de service. Voici l'application :

```
package javasoap.livre.ch4;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
public class AppAppelleCompteur {
   public static void main( String[] args )
      throws Exception {

      HTTP.startup("http://localhost:8004/glue");
      Context contexte = new Context();
      contexte.addProperty("activation", "application");
      Registry.publish( "urn:ServiceCompteurAppels",
         javasoap.livre.ch4.CompteurMethode.class, contexte );
   }
}
```

La classe `AppAppelleCompteur` contient la méthode `static main()` requise pour pouvoir être appelée en ligne de commande. La méthode `HTTP.startup()` démarre une instance d'un serveur HTTP sur ma machine locale `localhost` sur le port 8004 et la ressource utilisée pour accepter et router les messages SOAP est `/glue`. Tout cela mis ensemble nous donne l'URL complète *http://localhost:8004/glue*, envoyée comme paramètre à `HTTP.startup()`.

Après avoir démarré le serveur, nous devons y enregistrer notre service. Nous commençons par spécifier le modèle d'activation que nous souhaitons utiliser. Pour cela, nous créons une instance de la classe `electric.util.Context`. Cette classe est un composant de l'API GLUE et elle est utilisée pour assembler une collection de nom de propriétés et de valeurs associées. Pour enregistrer le modèle d'activation, nous appelons la méthode `addProperty()` de l'objet `contexte` en lui fournissant le nom de propriété `activation` et la valeur associée `application`. Nous sommes prêts à publier le service, en utilisant la méthode `publish()` de la classe GLUE `electric.registry.Registry`. Le premier paramètre passé est `urn:ServiceCompteurAppels` qui une fois encore est le nom du service à publier. Le second paramètre est la classe Java qui implémente le service, `CompteurMethode`. L'objet `contexte` est passé en dernier paramètre, indiquant que nous voulons publier (déployer) le service en utilisant le modèle d'activation de niveau application.

Pour démarrer l'application, nous entrons ce qui suit en ligne de commande :

```
java javasoap.livre.ch4.AppAppelleCompteur
```

Nous voyons alors en sortie le numéro de version GLUE ainsi que que l'URL complète de la ressource GLUE à l'écoute des messages SOAP. Il s'agit de la même URL que celle passée à la méthode `HTTP.startup()`. À ce stade, le `service urn:ServiceCompteurAppels` est déployé et prêt à accepter les invocations de méthodes RPC SOAP.

Comment GLUE peut-il savoir quelles méthodes sont exposées par le service ? Dans le descripteur de déploiement Apache SOAP, nous avions spécifiquement défini les méthodes exposées. Nous n'avons rien fait de tel sous GLUE. À la place, GLUE utilise plusieurs techniques pour déterminer automatiquement les méthodes. Par conséquent, la façon la plus simple de déployer un service est de laisser GLUE trouver par lui-même les méthodes à exposer. Aux chapitres suivants, nous verrons comment contrôler les méthodes exposées. Dans cet exemple, GLUE expose toutes les méthodes publiques de la classe `CompteurMethode` à l'exception du constructeur. Vous ne serez pas surpris de voir que GLUE requiert un constructeur sans arguments, tout comme Apache SOAP.

La console GLUE

GLUE contient une interface web permettant de voir les services disponibles sur un serveur donné, de les éditer et même d'en déployer de nouveaux. La console GLUE est une application web spéciale qui peut être démarrée par le script *console.bat* (ou *console.sh*) inclus sous le répertoire */bin* de l'installation GLUE. Si vous utilisez la ligne de commande sous Microsoft Windows, je vous suggère de vous rendre sous ce répertoire pour exécuter la commande `console`, car `console` est également une commande Windows. Si vous vous positionnez préalablement sous le répertoire */bin* de GLUE, vous démarrerez le bon programme.

Par défaut, la console GLUE écoute les connexions HTTP sur le port 8100 de la machine locale (i.e., *http://localhost:8100*). Pour exécuter l'application sur un autre port, spécifiez le numéro de port comme argument de la commande `console`. La commande suivante démarre la console sur le port 8150 :

```
console 8150
```

Tous les exemples de ce livre utilisent la port par défaut 8100. Démarrez la console puis lancez votre navigateur web et allez sur *http://localhost:8100*. Vous verrez la fenêtre principale de la console GLUE comme dans la *figure 4-5*.

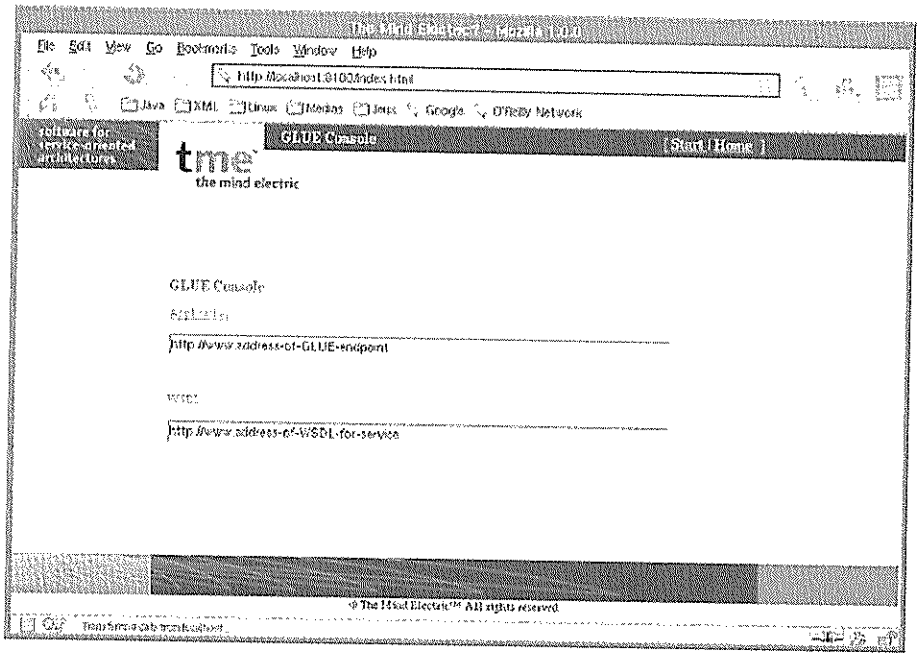


Figure 4-5. La console GLUE

Nous n'allons pour l'instant utiliser que le premier champ de l'interface de la console GLUE, dont le label est HOME.[7] Souvenez-vous que nous avons déjà démarré une instance de notre serveur GLUE à l'aide de l'URL *http://localhost:8004/glue*. Entrez cette URL dans le champ HOME de l'écran de console, cliquez sur le bouton HOME et vous serez dirigé vers la page que l'on voit *figure 4-6*. Cette page inclut l'URL du serveur de destination qui est la ressource qui gère les messages SOAP. Elle montre également tous les services actuellement disponibles sur ce serveur. Le dernier, `system/admin`, est un service spécial utilisé par GLUE. Le premier service est `urn:ServiceCompteurAppels`, que nous avons précédemment déployé. Le lien `urn:ServiceCompteurAppels` vous emmène sur une page décrivant le service web, comme on le voit à la *figure 4-7*.

7. NdR : Dans Glue 3.1 le label est Application.

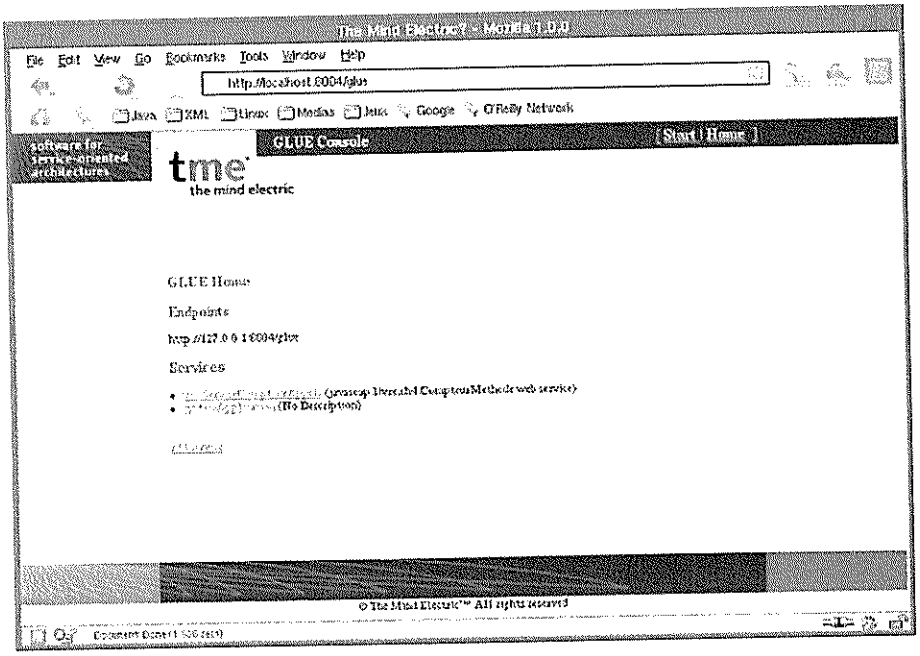


Figure 4-6. La page d'accueil GLUE

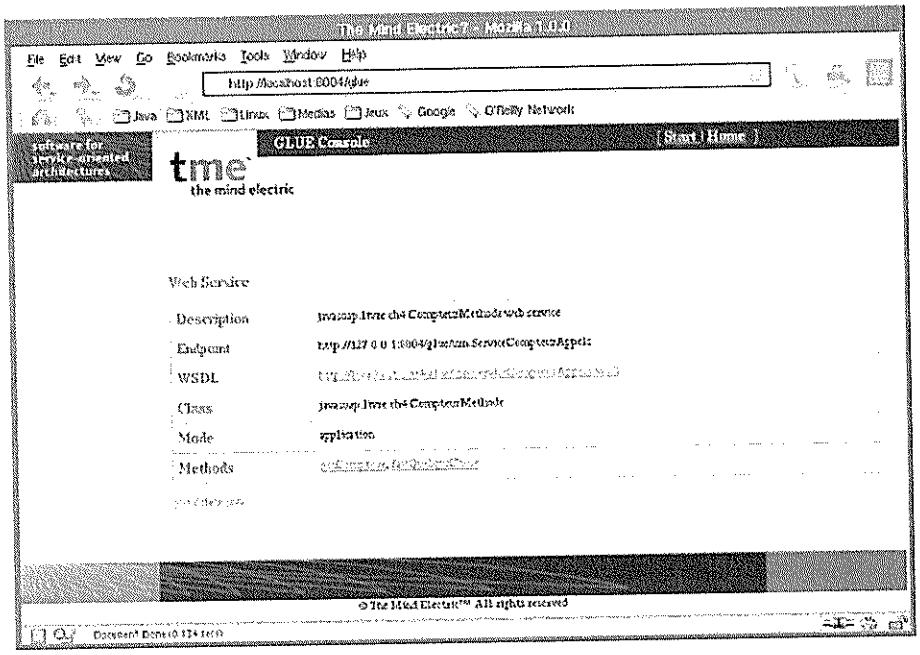


Figure 4-7. Informations sur le service Web

Examinons les informations fournies pour un service. Le champ Description donne la description du service. Sa valeur est une valeur par défaut générée pour nous par GLUE. Dans les exemples suivants, nous fournirons une description plus utile. Le champ Endpoint contient l'URL complète du service. Le champ WSDL présente une URL qui peut être utilisée pour demander une description du service en Web Service Description Language. Nous verrons cela au chapitre 9. Le champ Class montre le nom complet de la classe qui implémente le service et le champ Mode montre que le modèle d'activation utilisé est `application`. Enfin, le champ Methods contient les méthodes qui peuvent être invoquées sur ce service. Ceci est automatiquement déterminé par GLUE.

Le bouton EDIT vous emmène sur un écran qui vous permet d'éditer la description, la classe Java et le mode (modèle d'activation de service) du service. Il vous permet également de modifier l'URN (universal resource name). L'URN est le nom réel du service, qui est dans notre cas `urn:ServiceCompteurAppels`. Cliquer sur le bouton JAVA vous donne une liste des classes Java qui peuvent être utilisées côté client pour accéder au service. Nous aborderons cela à la section *Appel de service sous GLUE*.

Vous pouvez supprimer le service en cliquant sur le bouton Delete. En réponse, la console GLUE affiche la page de la *figure 4-8*.

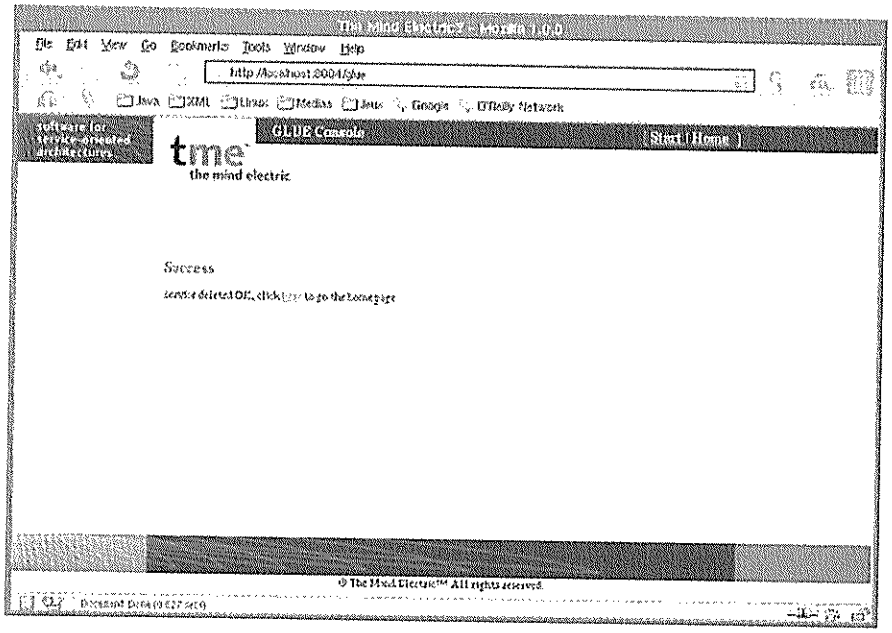


Figure 4-8. Suppression réussie

À ce stade, cliquez sur le lien « here » » pour retourner à la page décrivant les services web disponibles sur le serveur. Le résultat ressemblera à celui de la *figure 4-6* mais `urn:ServiceCompteurAppels` ne sera plus listé. Cliquez maintenant sur le bouton ADD, qui vous emmène sur la page où figurent les informations à fournir pour déployer un service (*figure 4-9*). J'ai entré ces informations pour `urn:ServiceCompteurAppels`, en donnant cette fois une description plus précise. J'ai utilisé les mêmes URN, classe et mode que lors du déploiement par l'application Java.

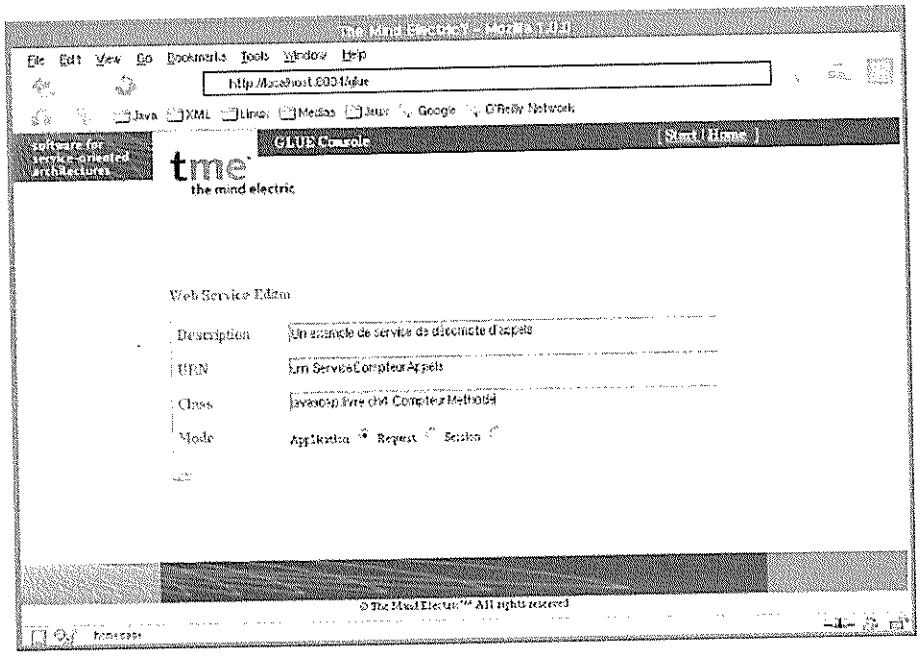


Figure 4-9. Ajout d'un service à partir de la console

Pressez le bouton SAVE pour déployer le service. Si vous avez fourni un nom de classe invalide, la console GLUE indiquera un échec. Vous serez sinon dirigé vers la page du service web (*figure 4-7*).

Écriture de clients pour les services

Dans les sections suivantes, nous allons écrire du code Java invoquant des méthodes de services exposées aussi bien par Apache SOAP que par GLUE. Pour l'instant, nous ne mélangerons pas les technologies. Nous allons utiliser les API Apache SOAP pour accéder à un serveur Apache SOAP et les API GLUE pour accéder à un serveur GLUE.

Qu'en est-il alors de l'interopérabilité ? L'un des plus importants aspects de SOAP en tant que protocole de niveau transmission, est que votre choix d'implémentation ne doit pas vous empêcher de communiquer avec succès avec d'autres implémentations SOAP. Il subsiste cependant encore quelques problèmes quant à l'interopérabilité SOAP. Certaines technologies y sont actuellement meilleures que d'autres et dans certains cas, vous aurez à traverser quelques épreuves avant de faire communiquer correctement des technologies SOAP différentes. Laissons donc ce sujet de côté jusqu'au chapitre 9, dans lequel nous traiterons ces questions en détail.

Appel de service sous Apache SOAP

Maintenant que nous avons un service opérationnel, écrivons le code Java qui agira comme un client dudit service. Voici une application Java très simple qui invoque la méthode `getCompteur` d'`urn:ServiceCompteurAppels` sur notre serveur Apache SOAP :

```
package javasoap.livre.ch4;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class AppRenvoieCompteur {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");

      Call appel = new Call();
      appel.setTargetObjectURI("urn:ServiceCompteurAppels");
      appel.setMethodName("getCompteur");
      try {
         Response rep = appel.invoke(url, "");
         Parameter ret = rep.getReturnValue();
         Object valeur = ret.getValue();
         System.out.println("Le résultat est " + valeur);
      }
      catch (SOAPException e) {
         System.err.println("SOAPException (" +
                        e.getFaultCode() + "): " +
                        e.getMessage());
      }
   }
}
```

Ce programme commence par créer une instance de `java.net.URL` en utilisant l'URL du routeur RPC Apache SOAP de notre serveur. Il crée ensuite une instance de `org.apache.soap.rpc.Call` qui est une classe Java encapsulant un appel de méthode RPC SOAP. La méthode `setTargetObjectURI()` de l'objet `appel` est utilisée pour spécifier le nom du service et `setMethodName()` indique le nom de la méthode invoquée.

Arrivés à ce stade, nous sommes prêts à invoquer notre méthode sur le service spécifié. Dans la mesure ou cette invocation est susceptible de retourner une `SOAPException`, nous l'avons encapsulé dans un bloc `try/catch`. Si quoi que ce soit se déroule mal, nous afficherons les code et message de faute de l'exception.

Pour effectuer l'appel, nous utilisons la méthode `invoke()` de l'objet `appel`, en lui passant comme premier paramètre l'URL précédemment instanciée. Le second paramètre est l'URI d'une action SOAP, incluse comme une entrée d'en-tête HTTP. Nous n'utilisons pas d'action SOAP et une chaîne vide est donc passée en paramètre. La méthode `invoke()` renvoie une instance de `org.apache.soap.rpc.Response`. En cas de réussite, cet objet contiendra la réponse du serveur. Nous supposerons pour l'instant que l'appel a été couronné de succès. Après avoir invoqué la méthode, nous appelons la méthode `getReturnValue()` de l'objet réponse. Cette méthode renvoie une instance de `org.apache.soap.rpc.Parameter`, contenant la valeur de retour ou la réponse à l'appel de méthode du service. Nous appelons maintenant la méthode `getValue()` du paramètre, qui renvoie une instance d'`Object`. Le type réel de l'objet ainsi récupéré dépend du type re-

tourné par la méthode. Ceci est un point important. Les appels de méthodes RPC Apache SOAP renvoient des instances d'objets Java, non des instances de primitives Java. Dans un cas tel que celui-ci où la méthode de service appelée renvoie une valeur `int`, la valeur primitive de retour est créée sous la forme d'une instance du type de son objet Java correspondant, `Integer` dans ce cas. Nous pourrions caster la valeur de retour en `Integer` mais au lieu de cela, nous la conservons sous forme d'une instance d'`Object` et affichons le résultat à l'aide de `System.out.println()`. Cette méthode appelle implicitement `toString()`, qui renvoie la valeur `Integer` sous forme de `String`.

Exécutons cette application :

```
java javasoap.livre.ch4.AppRenvoieCompteur
```

Si tout a été correctement préparé, la sortie suivante doit s'afficher :

```
Le résultat est 0
```

Le résultat est 0 car nous n'avons effectué pour l'instant aucun appel à la méthode `faitQuelqueChose()` qui incrémente la valeur du compteur de l'objet du service. Merveilleux ! Nous avons déployé et invoqué avec succès en Java un service SOAP. Voyons la charge utile SOAP utilisée pour invoquer le service :

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: Localhost
Content-Type: text/xml; charset=iso-8859-1
Content-Length: 344
SOAPAction: ""

<?xml version='1.0' encoding='iso-8859-1'?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <SOAP-ENV:Body>
  <ns1:getCompteur xmlns:ns1="urn:ServiceCompteurAppels">
  </ns1:getCompteur>
 </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Toutes les données de ce document SOAP doivent vous sembler familières. Les informations spécifiques à cet exemple sont contenues dans le corps SOAP, dans lequel vous pouvez voir que l'élément `getCompteur` (qui représente la méthode invoquée) est qualifié par l'espace de noms `urn:ServiceCompteurAppels` qui est le nom du service dont la méthode est invoquée. Pendant que nous y sommes, examinons la réponse renvoyée par le serveur :

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=iso-8859-1
Content-Length: 468
Set-Cookie2: JSESSIONID=yqf5f6su31;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=yqf5f6su31;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.3
```

```
<?xml version='1.0' encoding='iso-8859-1'?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <SOAP-ENV:Body>
  <ns1:renvoieReponseCompteur
   xmlns:ns1="urn:ServiceCompteurAppels"
   SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
   <return xsi:type="xsd:int">0</return>
  </ns1:renvoieReponseCompteur>
 </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Les en-têtes HTTP sont affichés afin que vous puissiez voir l'intégralité de la réponse. Vous noterez que le serveur renvoie des cookies HTTP. Bien que nous ne les utilisions pas encore, ces cookies seront utiles lorsque nous utiliserons l'activation de services au niveau `Session`.

Voyons maintenant le corps SOAP. Le nom de la méthode invoquée étant `getCompteur`, le premier élément enfant du corps est `renvoieReponseCompteur`, qualifié par le nom de service. L'attribut`SOAP-ENV:encodingStyle` est positionné à `http://schemas.xmlsoap.org/soap/encoding`. Dans l'élément `renvoieReponseCompteur`, nous trouvons un élément enfant simple nommé `return` contenant la valeur de retour de la méthode `getCompteur()`. `return` est explicitement typé en `xsd:int` et contient la valeur `0`.

Effectuons quelques appels à `faitQuelqueChose` pour incrémenter le compteur :

```
package javasoap.livre.ch4;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class AppRenvoieCompteur2 {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");

      Call appel = new Call();
      appel.setTargetObjectURI("urn:ServiceCompteurAppels");
      try {
         appel.setMethodName("faitQuelqueChose");
         Response rep = appel.invoke(url, "");
         rep = appel.invoke(url, "");
         rep = appel.invoke(url, "");
         appel.setMethodName("getCompteur");
         rep = appel.invoke(url, "");
```

```
            Parameter ret = rep.getReturnValue();
            Object valeur = ret.getValue();
            System.out.println("Le résultat est " + valeur);
        }
        catch (SOAPException e) {
            System.err.println("SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

Dans cet exemple, nous avons d'abord positionné le nom de la méthode à `faitQuelqueChose` en utilisant `setMethodName()` puis exécuté `appel.invoke()` trois fois. Le nom de méthode est ensuite positionné à `getCompteur` et le service est de nouveau invoqué. Comme la fois précédente, le résultat de la méthode `getCompteur` est affiché. La sortie suivante devrait s'afficher :

```
Le résultat est 3
```

Nous avons déployé `urn:ServiceCompteurAppels` en utilisant une portée de type application, signifiant qu'une même instance de la classe du service est utilisée à chaque requête. De ce fait, à chaque appel de la méthode `faitQuelqueChose`, la variable `compteur` de l'instance de `CompteurMethode` est incrémentée. Si vous exécutez l'application de nouveau, le résultat sera 6 et une exécution supplémentaire donnera 9. Vous voyez ici la portée de type application en action, aussi trivial puisse être l'exemple utilisé. Si vous voulez réinitialiser le compteur à 0, arrêtez le serveur hébergeant Apache SOAP et redémarrez-le.

Vous pourriez penser qu'une alternative à l'arrêt/redémarrage du serveur pourrait être de supprimer puis de redéployer le service. Il n'en est rien, en tout cas avec le serveur Tomcat. Il semble que la classe implémentant le service est mise en cache et réutilisée si le service est redéployé. Si vous utilisez un autre serveur pour héberger Apache SOAP, vous devrez vérifier ce point par vous-même.

Appel de service sous GLUE

Créons maintenant une application utilisant les API GLUE pour communiquer avec le serveur GLUE que nous avons démarré précédemment. GLUE aborde différemment la programmation de clients Java. Il permet d'accéder à des services *via* des interfaces Java locales. Ces interfaces contiennent des méthodes correspondant aux méthodes Java du service exposées ou publiées. Dans cet exemple, nous allons écrire une interface Java possédant les méthodes `getCompteur()` et `faitQuelqueChose()` telles qu'elles apparaissent dans `CompteurMethode`. Voici une interface de ce type, nommée `IServiceCompteurAppels`.[8]

8. Faire débuter les noms d'interfaces avec un I est une pratique courante mais n'est en aucun cas une convention Java !

```
package javasoap.livre.ch4;
public interface IServiceCompteurAppels {
  int getCompteur();
  boolean faitQuelqueChose();
}
```

Créons ensuite une application accédant au service `urn:ServiceCompteurAppels` hébergé sous GLUE.

```
package javasoap.livre.ch4;
import electric.registry.Registry;
import electric.registry.RegistryException;
public class AppRenvoieCompteur3 {
   public static void main(String[] args) throws Exception
   {
      try {
         IServiceCompteurAppels ctr =
           (IServiceCompteurAppels)Registry.bind(
             "http://localhost:8004/glue/urn:ServiceCompteurAppels.wsdl",
              IServiceCompteurAppels.class);
       System.out.println("Le résultat est " + ctr.getCompteur());
     }
     catch (RegistryException e)
     {
        System.out.println(e);
     }
   }
}
```

GLUE nous permet de lier l'interface au service. Une fois ceci effectué, nous pouvons utiliser l'interface comme si elle était un objet local (ce qu'elle est). Pour cela, la seule chose à faire est d'appeler la méthode statique `bind()` de la classe `electric.registry.Registry`. Le premier paramètre est une URL vers un fichier WSDL décrivant le service. GLUE est basé sur WSDL et génère automatiquement une description WSDL du service. (WSDL sera brièvement abordé au chapitre 9.) Pour créer l'URL appropriée, prenez l'adresse du serveur et de la ressource (*http://localhost:8004/glue*) et ajoutez-y le nom du service, suivi de *.wsdl*. Le second paramètre est la classe Java représentant l'interface, `IServiceCompteurAppels.class`. La valeur de retour de la méthode `Registry.bind()` est castée en une instance de l'interface Java pour le service, `IServiceCompteurAppels`.

Une fois le retour de l'appel à `Registry.bind()` obtenu, nous pouvons utiliser la variable `ctr` pour appeler les méthodes `getCompteur()` et `faitQuelqueChose()` comme si elles n'étaient guère plus qu'un objet Java local. Dans cet exemple, nous affichons simplement le résultat en appelant `ctr.getCompteur()`. Voici comment s'exécute l'exemple :

```
java javasoap.livre.ch4.AppRenvoieCompteur3
```

Le résultat de l'appel à `getCompteur()` est 0 car il n'y a pas encore eu d'appel à `faitQuelqueChose()` :

```
Le résultat est 0
```

La console GLUE peut générer du code Java rendant plus facile l'accès à un service. Pour générer ce code, revenez sur la console GLUE avec votre navigateur puis allez sur `urn:ServiceCompteurAppels` et cliquez sur le bouton JAVA. GLUE utilise un nom pour l'interface (`ICompteurMethode`) différent du mien mais le contenu lui-même reste identique. La classe technique est nommée `CompteurMethodeTechnique`. Elle encapsule les appels à `Registry.bind()` ainsi que le cast requis pour passer la bonne interface. La classe technique fournit deux méthodes statiques `bind()`. La première n'attend pas de paramètres et génère automatiquement l'URL du service visé. L'autre `bind()` permet de spécifier une autre URL si besoin est. Voici l'application cliente précédente modifiée pour utiliser l'interface Java et la classe technique générée par la console. Assurez-vous d'utiliser le nom de package approprié pour l'interface et la classe technique. La console GLUE n'effectue pas cette portion du travail et elle vous revient donc.[9]

```
package javasoap.livre.ch4;
import electric.registry.RegistryException;
public class AppRenvoieCompteur4 {
   public static void main(String[] args) throws Exception
   {
      try {
       ICompteurMethode ctr = CompteurMethodeTechnique.bind();
       System.out.println("Le résultat est " + ctr.getCompteur());
     }
     catch (RegistryException e)
     {
        System.out.println(e);
     }
   }
}
```

L'utilisation de la classe technique `CompteurMethodeTechnique` clarifie un peu le code. Pour un court programme comme celui-ci, la différence n'est pas flagrante mais pour une application plus conséquente, la rentabilité serait notable. Compilez le nouveau client avec l'interface Java et la classe technique incorporées au package `javasoap.livre.ch4`. L'exécution de cet exemple depuis la ligne de commande nous donne le même résultat qu'avant.

La console GLUE donne également un moyen d'invoquer les méthodes d'un service sans écrire la moindre ligne de code. Revenez dans votre navigateur sur la page du service web `urn:ServiceCompteurAppels`. Les méthodes listées dans le champ METHODS sont des hyperliens. Cliquer sur l'un d'entre eux vous dirige vers une page permettant d'invoquer la méthode de service sélectionnée. En sélectionnant la méthode `getCompteur`, vous verrez la page montrée *figure 4-10*. On y voit deux champs destinés aux paramètres d'entrée et aux valeurs de retour, INPUTS et OUTPUTS. Si des paramètres étaient attendus de la méthode `getCompteur()`, les valeurs seraient entrées à cet endroit. Il n'y en a pas dans notre cas. Cliquer sur le bouton SEND invoque la méthode du service. Au retour de l'appel, le résultat est affiché en dessous du bouton. Si vous l'effectuez

9. GLUE fournit un utilitaire nommé `wsdl2java` qui génère du code Java à partir d'un fichier WSDL et offre en option de pouvoir générer le code d'un package spécifié. Nous verrons cet utilitaire au chapitre 9.

maintenant, vous verrez un 0. Cette interface web fournit un outil très utile pour vérifier les fonctionalités de services web sans avoir à écrire de code Java.

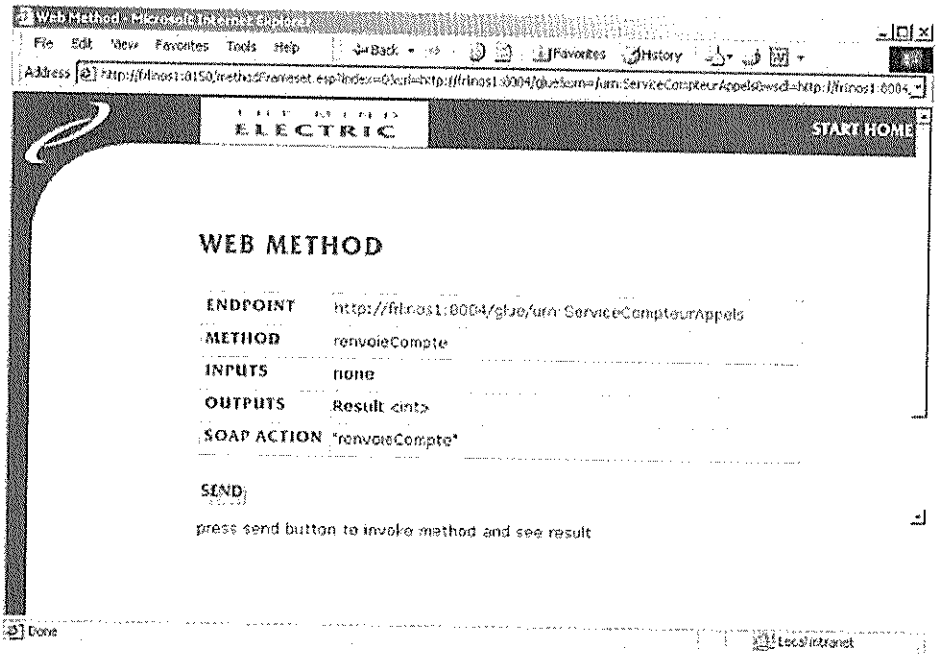


Figure 4-10. Invocation de getCompteur depuis un navigateur

Modifions l'exemple GLUE afin d'effectuer de multiples appels à `faitQuelqueChose` avant d'appeler `getCompteur()`. Voici une nouvelle version de notre client qui utilise les API GLUE pour effectuer plusieurs appels ainsi que l'interface et la classe technique générées par la console GLUE :

```
package javasoap.livre.ch4;
import electric.registry.RegistryException;
public class AppFaitQuelqueChose {
   public static void main(String[] args) throws Exception
   {
      try {
       ICompteurMethode ctr = CompteurMethodeTechnique.bind();
       ctr.faitQuelqueChose();
       ctr.faitQuelqueChose();
       ctr.faitQuelqueChose();
       System.out.println("Le résultat est " + ctr.getCompteur());
    }
    catch (RegistryException e)
    {
       System.out.println(e);
    }
  }
}
```

Lorsque vous exécuterez ce client pour la première fois, vous obtiendrez :

```
Le résultat est 3
```

Si vous l'exécutez à nouveau, le résultat sera 6 et ainsi de suite, car le service utilise une portée de type application (mode d'activation). Pour réinitialiser le compteur à 0, arrêtez et redémarrez l'application `AppAppelleCompteur` ou utilisez la console GLUE pour supprimer le service et le recréer. Sous GLUE, ceci fonctionne correctement.

Déploiement avec une portée de niveau requête

Jusqu'à maintenant, tous les exemples que nous avons créé ont été déployés en utilisant une portée de niveau application (activation de service). Dans ce mode, une seule instance de la classe d'implémentation du service traitait toutes les invocations de méthodes. Modifions le déploiement et utilisons la portée de niveau requête. Avec cette dernière, une nouvelle instance de l'objet du service est créée à chaque invocation de méthode et cet objet est détruit une fois l'invocation terminée.

Nous allons tout d'abord modifier l'exemple fonctionnant sous Apache SOAP en changeant son descripteur de déploiement. La seule modification requise est de positionner l'attribut `scope` de l'élément `provider` à `Request`. Le reste du descripteur de déploiement reste exactement le même :

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:ServiceCompteurAppels">

   <isd:provider type="java"
         scope="Request"
         methods="faitQuelqueChose getCompteur">
     <isd:java class="javasoap.livre.ch4.CompteurMethode"
         static="false"/>
   </isd:provider>
   <isd:faultListener>
   org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
   <isd:mappings>
   </isd:mappings>
</isd:service>
```

N'oubliez pas de supprimer et redéployer `urn:ServiceCompteurAppels` à l'aide d'une des techniques évoquées plus haut. Vous n'avez pas besoin d'arrêter le serveur comme nous l'avons fait précédemment. Maintenant que nous utilisons la portée de niveau requête, il ne devrait plus y avoir d'instance de `CompteurMethode` en mémoire. Vous pouvez donc utiliser la version en ligne de commande du client gestionnaire de service pour supprimer et redéployer le service ou l'application d'administration d'Apache SOAP depuis votre navigateur. Si vous décidez d'arrêter et de redémarrer le serveur, vous devrez quand même supprimer et redéployer le service car Apache SOAP maintient de façon persistante les services déployés.

Maintenant que vous avez redéployé `urn:ServiceCompteurAppels` en utilisant une portée de niveau requête, vous pouvez utiliser la version Apache de l'application cliente pour accéder au service. Bien que l'application appelle `faitQuelqueChose()` plusieurs

fois avant d'appeler `getCompteur()`, le résultat sera toujours 0. Ceci est une conséquence directe de l'utilisation de la portée de niveau requête : à chaque fois que l'une de ces méthodes est invoquée, une nouvelle instance de la classe `CompteurMethode` est créée et cette instance est détruite une fois l'invocation terminée. Aucun état n'est maintenu entre deux invocations. De ce fait, il n'existe aucun moyen d'obtenir un résultat autre que 0.

Faisons la même chose avec le service s'exécutant sous GLUE. La façon la plus simple de procéder est d'utiliser la console GLUE. Affichez la page des services pour `urn:ServiceCompteurAppels` et cliquez sur le bouton EDIT. Cela vous dirigera sur la page d'édition de service web que l'on voit *figure 4-11*.

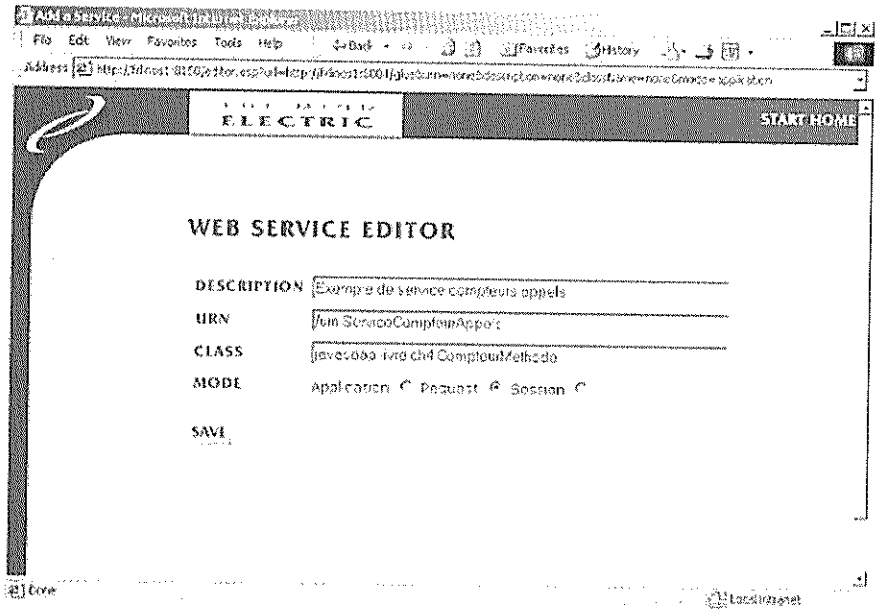


Figure 4-11. L'éditeur de service web

Pour modifier le mode d'activation, cliquez sur le bouton radio correspondant dans le champ MODE et sélectionnez Request. Cliquez maintenant sur le bouton SAVE et le service se trouvera déployé avec une portée de niveau requête.

J'ai utilisé plusieurs termes pour le modèle d'activation de service. Je l'appelle parfois la portée, parfois le mode d'activation et parfois encore le modèle d'activation de service. Ils sont synonymes mais les implémentations utilisent des termes différents et je préfère vous y habituer. Les mots peuvent être différents, ils se réfèrent tous à la durée de vie de l'objet qui implémente le service.

Vous pouvez vérifier que le service utilise maintenant une portée de niveau requête en exécutant de nouveau l'application `AppFaitQuelqueChose`. Vous obtiendrez toujours le résultat 0. Encore une fois, les appels à `faitQuelqueChose` et l'appel à `getCompteur` sont tous faits à des instances distinctes de la classe `CompteurMethode`.

Déploiement avec une portée de niveau session

Apache SOAP pourvoie à la gestion de sessions en envoyant des cookies *via* les en-têtes HTTP. Plus haut dans ce chapitre, nous avons vu une réponse HTTP du serveur, incluant des entrées d'en-tête HTTP nommées `Set-Cookie` et `Set-Cookie2`. L'application cliente utilise ces cookies si elle doit effectuer des invocations successives de méthodes dans la même session que la requête d'origine. L'API Apache SOAP API utilise une technique simple pour gérer cela. `org.apache.soap.transport.http.SOAPHTTPConnection` inclut une méthode nommée `setMaintainSession()` recevant un simple booléen en paramètre. Ce paramètre, lorsque lui est affectée la valeur `true`, demande à l'objet utilisé par la connexion de maintenir la session en cours. L'objet utilisé par la connexion implémente cette demande en gardant une trace des cookies et en les renvoyant au serveur lorsque l'invocation suivante est effectuée.

Éditons le descripteur de déploiement et positionnons l'attribut `scope` à `Session`. Exécutons à nouveau le programme client. On obtient le résultat suivant :

```
Le résultat est 3
```

Ceci prouve que la même instance de l'objet du service a été utilisée pour toutes les invocations de méthode. Si vous l'exécutez de nouveau, vous obtiendrez le même résultat car une nouvelle instance de l'objet destiné à la session est créée au premier appel à `faitQuelqueChose()` et que chaque appel successif à la méthode utilise le même objet de service. Comme nous utilisons la même instance de `org.apache.soap.rpc.Call` à chaque appel au service, nous utilisons également la même instance de `org.apache.soap.transport.http.SOAPHTTPConnection`. Vous avez probablement deviné que le comportement par défaut de `org.apache.soap.transport.http.SOAPHTTPConnection` est de maintenir la session.

La portée au niveau de la session est employée lors de l'utilisation de chariots d'achats, lorsque l'on veut garder une trace de la session entre deux invocations de méthode. Cependant, il peut y avoir des cas dans lesquels vous ne voudrez pas utiliser la même session pour tous vos appels de méthode, même lorsque le service est déployé avec une portée niveau session. Dans l'exemple suivant, l'objet `SOAPHTTPConnection` est récupéré de l'objet `Call` après la première invocation à la méthode. Puis un appel à la méthode `setMaintainSession()` de la connexion est effectué avec le paramètre `false`. Ceci fait que la connexion cesse d'utiliser la même session lors d'appels de méthodes successifs.

```
package javasoap.livre.ch4;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.transport.http.*;
public class AppSession {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");

      Call appel = new Call();
      appel.setTargetObjectURI("urn:ServiceCompteurAppels");
```

```
        try {
           appel.setMethodName("faitQuelqueChose");
           Response rep = appel.invoke(url, "");
           SOAPHTTPConnection st =
            (SOAPHTTPConnection)appel.getSOAPTransport();
           st.setMaintainSession(false);
           rep = appel.invoke(url, "");
           st.setMaintainSession(true);
           rep = appel.invoke(url, "");
           appel.setMethodName("getCompteur");
           rep = appel.invoke(url, "");
           Parameter ret = rep.getReturnValue();
           Object value = ret.getValue();
           System.out.println("Le résultat est " + value);
        }
        catch (SOAPException e) {
           System.err.println("SOAPException (" +
                              e.getFaultCode() + "): " +
                              e.getMessage());
        }
     }
  }
```

Avant le troisième appel à `faitQuelqueChose()`, nous appelons `setMaintainSession()` de nouveau avec le paramètre `true`. De ce fait, après le troisième appel à `faitQuelqueChose()`, l'objet `SOAPHTTPConnection` gardera trace des cookies de la session. Et lorsque nous appellerons `getCompteur()`, ces cookies seront renvoyés au serveur, assurant ainsi que le même objet de session sera de nouveau utilisé. Vous devriez obtenir le résultat suivant lorsque vous exécutez l'application `AppSession` :

```
Le résultat est 1
```

Si vous voulez maintenir une seule session au travers d'appels multiples, assurez-vous que vous établissez la session avant le premier appel qui devra en faire partie. Programmez ainsi est cependant un peu un comportement de fainéant, du moins à mon goût. Je préférerais de loin créer une nouvelle instance de `SOAPHTTPConnection` et l'utiliser lorsque je commencerai à effectuer des appels utilisant la même session. Ces considérations ne s'appliquent naturellement qu'à des services déployés avec une portée de niveau session. Si vous voulez essayer, modifiez le code situé à l'intérieur du bloc `try` de la façon suivante :

```
appel.setMethodName("faitQuelqueChose");
Response rep = appel.invoke(url, "");
rep = appel.invoke(url, "");
SOAPHTTPConnection st = new SOAPHTTPConnection();
appel.setSOAPTransport(st);
rep = appel.invoke(url, "");
appel.setMethodName("getCompteur");
rep = appel.invoke(url, "");
Parameter ret = rep.getReturnValue();
Object value = ret.getValue();
System.out.println("Le résultat est " + value);
```

La durée de vie d'une session est dépendante de l'implémentation. Le serveur que vous utilisez, la façon dont il est configuré et les différentes options possibles jouent tous un rôle dans la durée d'une session. La plupart du temps, le serveur termine une session au bout d'une certaine période d'inactivité. GLUE gère automatiquement les sessions et vous laisse régler la durée de vie des sessions en appelant `electric.server.http.HTTP.setSessionTimeout()`. Le paramètre passé à cette méthode est un entier contenant la durée en secondes pendant laquelle une session peut rester inactive avant son invalidation.

Passage de paramètres

Jusqu'ici, les exemples de ce chapitre ont utilisé des méthodes ne contenant aucun paramètre. Nous allons maintenant passer un peu de temps à étudier comment sont passés les paramètres. Créons un service nommé `urn:ServicePrixActions` et implémenté par la classe `javasoap.livre.ch4.PrixActions`. La méthode `getPrix()` reçoit en paramètres le symbole de l'action ainsi que la devise. La méthode renvoie toujours la valeur de type flottant 75.33. Pas très intéressante comme action mais celle-là au moins, elle ne baissera pas. Plus sérieusement, nous sommes surtout intéressés par la façon de passer les paramètres. Les détails concernant la façon d'accéder à une base de données pour trouver le prix d'une action, sont pour le moment de votre ressort.

```
package javasoap.livre.ch4;
public class PrixActions {
   public float getPrix(String action, String devise) {
      float resultat;
      // détermine le prix d'une action et le renvoie
      // dans la devise spécifiée
      resultat = (float)75.33;
      return resultat;
   }
}
```

Nous ne nous étendrons pas sur les détails du déploiement, que ce soit pour Apache SOAP ou pour GLUE dans la mesure où nous avons déjà consacré l'essentiel de ce chapitre à ce sujet. Voici le descripteur de déploiement Apache SOAP pour le `ServicePrixActions` implémenté par la classe `PrixActions`. Le service utilise une portée de niveau application et contient une seule méthode nommée `getPrix` et qui trouve le prix d'échange actuel d'une action dans une devise spécifique.

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:ServicePrixActions">

   <isd:provider type="java"
         scope="Application"
         methods="getPrix">
     <isd:java class="javasoap.livre.ch4.PrixActions"
         static="false"/>
   </isd:provider>
   <isd:faultListener>
   org.apache.soap.server.DOMFaultListener
```

```
    </isd:faultListener>
    <isd:mappings>
    </isd:mappings>
</isd:service>
```

Ce service peut être déployé sous Apache SOAP en utilisant la page d'administration dans un navigateur ou en mettant de descripteur de déploiement dans un fichier et en utilisant le client gestionnaire de service. Nous utiliserons cette dernière approche. Le fichier contenant le descripteur de déploiement est *ServicePrixActions.dd*. En utilisant ce fichier, la commande suivante déploiera le service sur ma machine locale `localhost`:

```
java org.apache.soap.server.ServiceManagerClient
  http://localhost:8080/soap/servlet/rpcrouter deploy ServicePrixActions.dd
```

Voici une application simple qui utilise les API Apache SOAP pour accéder au `ServicePrixActions`:

```
package javasoap.livre.ch4;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class AppPrixActions {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");

      Call appel = new Call();
      appel.setTargetObjectURI("urn:ServicePrixActions");

      appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String action = "MINDSTRM";
      String devise = "USD";
      Vector params = new Vector();
      params.addElement(new Parameter("action",
                        String.class, action, null));
      params.addElement(new Parameter("devise",
                        String.class, devise, null));
      appel.setParams(params);

      try {
         appel.setMethodName("getPrix");
         Response rep = appel.invoke(url, "");
         Parameter ret = rep.getReturnValue();
         Object valeur = ret.getValue();
         System.out.println("Le prix est " + valeur);
      }
      catch (SOAPException e) {
         System.err.println("SOAPException (" +
                        e.getFaultCode() + "): " +
```

```
                    e.getMessage());
            }
        }
    }
```

Le début de cette application est similaire aux applications que nous avons créé plus haut avec l'API Apache SOAP. Le code pour la mise en place des paramètres débute par un appel à la méthode `setEncodingStyleURI()` de l'objet `Call`. Le paramètre envoyé à cette méthode est une chaîne de caractères contenant l'URI du style d'encodage à utiliser pour sérialiser les paramètres. Dans notre cas, le style d'encodage est fournit par `Constants.NS_URI_SOAP_ENC`, qui est une constante définie par la classe `org.apache.soap.Constants`. La valeur de cette constante est `"http://schemas.xmlsoap.org/soap/encoding/"`, qui est l'espace de noms SOAP standard pour l'encodage. La valeur par défaut du style d'encodage d'une instance de `Call` est `null` et vous devrez donc songer à appeler la méthode `setEncodingStyleURI()`. À défaut, le serveur ne comprendra pas le schéma d'encodage à utiliser pour dé-sérialiser les paramètres.

Nous positionnons ensuite deux variables `String` contenant le symbole de l'action[10] et la devise à utiliser lors de l'invocation de la méthode `getPrix()`. Créons maintenant une instance de `java.util.Vector`, qui contiendra les paramètres. Chaque paramètre est porté par une instance de `org.apache.soap.rpc.Parameter`. Le constructeur de cette classe attend le nom du paramètre, la classe de l'objet contenant la valeur du paramètre, l'objet contenant cette dernière et l'URI du style d'encodage pour le paramètre. En premier paramètre du constructeur de `org.apache.soap.rpc.Parameter`, nous passons `action`, qui est le nom utilisé comme premier paramètre de la méthode `getPrix` de la classe `PrixActions`. Le second paramètre est `String.class`, qui indique au constructeur que la valeur est passée comme une instance d'une `String`. Le troisième paramètre est une variable `String` contenant la valeur que nous voulons réellement passer : `"MINDSTRM"`. Nous envoyons `null` en dernier paramètre, indiquant ainsi qu'aucune URI de style d'encodage ne doit être incluse pour l'élément XML qui sera sérialisé pour ce paramètre. Souvenez-vous que nous avons déjà positionné l'URI du style d'encodage pour l'ensemble de l'appel lorsque nous avons appelé `call.setEncodingStyleURI()`. À moins que vous ne vouliez en utiliser un autre, envoyez `null`. L'instance de `org.apache.soap.rpc.Parameter` est ajoutée au vecteur par la méthode `params.addElement()`. Nous procédons de façon identique pour le paramètre `devise` et l'ajoutons également au vecteur `params`.

Pour finir, nous passons le vecteur contenant les paramètres destinés à l'objet `Call` en utilisant la méthode `appel.setParams()`. À partir de là, le reste du code est familier. La méthode de service `getPrix()` est invoquée et le résultat est affiché. Voici l'enveloppe SOAP à envoyer au serveur :

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

10. MINDSTRM n'est pas un vrai symbole d'action.

```
<SOAP-ENV:Body>
  <ns1:getPrix
     xmlns:ns1="urn:ServicePrixActions"
     SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
     <action xsi:type="xsd:string">MINDSTRM</action>
     <devise xsi:type="xsd:string">USD</devise>
  </ns1:getPrix>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Comme nous l'avons déjà vu, le nom de la méthode que nous appelons est utilisé comme élément enfant du Body SOAP, qualifié par un espace de nom qui est le nom du service lui-même. L'attribut `SOAP-ENV:encodingStyle` de l'élément `getPrix` s'applique à tous ses sous-éléments à moins que ceux-ci ne comportent eux-mêmes un attribut `SOAP-ENV:encodingStyle`. (C'est pour cela que nous avons passé `null` au constructeur `org.apache.soap.rpc.Parameter`.) Les deux paramètres sont sérialisés comme éléments enfants de `getPrix`. Chacun d'eux est explicitement typé à l'aide de l'attribut `xsi:type` à la valeur `xsd:string` et la valeur du paramètre est sérialisée en tant que valeur de l'élément du paramètre associé.

Examinons maintenant ce qui est requis pour envoyer des paramètres à l'aide des API GLUE. Nous avons déjà vu que GLUE liait les interfaces Java aux services et que les méthodes de ces interfaces étaient traitées comme n'importe quelle autre instance Java. Il n'y a donc rien de très profond quant au passage de paramètres dans des applications basées sur GLUE. Voyons néanmoins les différentes étapes ne serait-ce que pour insister sur ces concepts.

En supposant que notre application `AppelCompteurApp` s'exécute toujours, nous pouvons utiliser la console GLUE pour déployer le nouveau service sur cette instance du serveur. En cliquant sur le bouton ADD de la page d'accueil de votre serveur, vous obtenez une page vierge d'édition d'un service web. Remplissez les détails nécessaires comme le montre la *figure 4-12*.

Dans cette écran, j'ai donné une courte description du service ainsi que son URN (`urn:ServicePrixActions`). La classe Java implémentant le service est `PrixActions` et le mode d'activation du service est Application. Cliquez sur SAVE pour le déployer.

Nous avons maintenant besoin d'une interface Java à lier à ce service. Vous pouvez utiliser celle générée par la console GLUE si vous préférez. Pour cet exemple, nous utiliserons l'interface `IServicePrixActions`, détaillée ci-dessous. Dans les deux cas, souvenez-vous que les noms utilisés par la console GLUE lorsqu'elle génère des interfaces sont différents de ceux que nous utilisons lorsque nous les créons nous-mêmes.

```
package javasoap.livre.ch4;
public interface IServicePrixActions {
  float getPrix(String action, String devise);
}
```

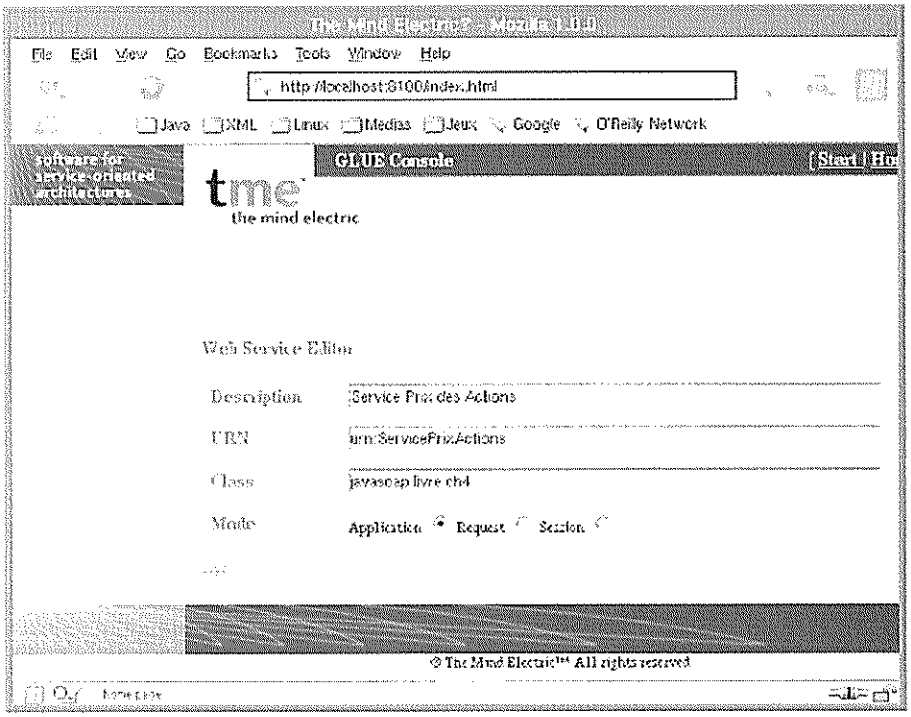


Figure 4-12. Spécification des détails dans l'éditeur de service web

L'écriture d'une application accédant à `ServicePrixActions` à l'aide des API GLUE est fort simple. Nous n'utiliserons pas cette fois la classe technique *via* la console GLUE bien qu'il n'y ait aucune raison de ne pas le faire. Voici l'application permettant d'accéder au service :

```
package javasoap.livre.ch4;
import electric.registry.Registry;
import electric.registry.RegistryException;
public class AppPrixActions2 {
   public static void main(String[] args) throws Exception
   {
      try {
         IServicePrixActions ctr =
           (IServicePrixActions)Registry.bind(
              "http://localhost:8004/glue/urn:ServicePrixActions.wsdl",
              IServicePrixActions.class);
         String action = "MINDSTRM";
         String devise = "USD";
         System.out.println("Le prix est " +
           ctr.getPrix(action, devise));
      }
      catch (RegistryException e)
      {
```

```
            System.out.println(e);
          }
        }
    }
```

Une fois une interface liée à un service, on peut l'employer directement comme si les objets du service étaient locaux. Tout ce dont nous avons besoin est d'appeler `ctr.getPrix()` en lui passant les variables `action` et `devise` en paramètres. Exécutez `AppPrixActions2` et vous obtiendrez le résultat suivant :

```
Le prix est 75.33
```

Nous n'avons pas encore examiné les messages SOAP transmis par des applications GLUE. Voyons comment ils diffèrent de ceux transmis par Apache SOAP. Voici l'enveloppe SOAP transmise par les API GLUE :

```
<soap:Envelope
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
 soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
  <soap:Body>
    <n:getPrix
       xmlns:n='http://tempuri.org/javasoap.livre.ch4.PrixActions'>
         <arg0 xsi:type='xsd:string'>MINDSTRM</arg0>
         <arg1 xsi:type='xsd:string'>USD</arg1>
    </n:getPrix>
  </soap:Body>
</soap:Envelope>
```

L'enveloppe GLUE générée par SOAP positionne les identifiants d'espaces de noms `xsi` et `xsd` de la même façon qu'Apache SOAP. Au lieu d'utiliser l'identifiant `SOAP-ENV`, GLUE utilise le nom `soap`, ce qui n'est guère une différence significative. Il s'agit juste d'un nom. Il peut être appelé n'importe comment du moment que lui est assignée la valeur `http://schemas.xmlsoap.org/soap/envelope`. L'attribut `encodingStyle` est également le même que dans les exemples Apache SOAP. Il semble qu'il y ait un attribut supplémentaire, `soapenc`, nullement référencé dans le message mais cela ne gênera en rien. L'élément `getPrix` qui représente la méthode de service n'est pas qualifié par un espace de noms utilisant le nom du service comme c'est la cas sous Apache SOAP. L'espace de noms est dérivé en ajoutant le nom de la classe d'implémentation du service à l'adresse *http://tempuri.org/*, qui est l'espace de noms par défaut généré par le serveur GLUE. Contrairement aux exemples Apache SOAP, les paramètres sont nommés `arg0`, `arg1`, etc. Dans la mesure où les paramètres de la méthode sont listés dans leur ordre d'apparition, leurs noms ne sont pas significatifs. Ils sont explicitement typés comme dans les exemples Apache.

Je préfère utiliser le nom réel du service pour qualifier le nom de sa méthode plutôt qu'un nom artificiel dérivé par défaut. Ceci est facile à corriger. Fournissez simplement une propriété espace de noms à l'objet contexte utilisé lorsque le serveur est exécuté. Nous avons procédé ainsi pour spécifier le mode d'activation du service. Voici une version modifiée de l'application serveur (originellement `AppAppelCompteur`), mais nous

avons spécifié cette fois l'espace de noms à utiliser en appelant `contexte.addProperty()` avec `"namespace"` en premier paramètre et `"urn:ServicePrixActions"` en second. Cette nouvelle application publie `urn:ServicePrixActions` et nous n'aurons donc pas à utiliser la console GLUE.

```
package javasoap.livre.ch4;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
public class AppPrixActions3 {
   public static void main( String[] args )
      throws Exception {

      HTTP.startup("http://localhost:8004/glue");
      Context contexte = new Context();
      contexte.addProperty("activation", "application");
      contexte.addProperty("namespace", "urn:ServicePrixActions");
      Registry.publish("urn:ServicePrixActions",
         javasoap.livre.ch4.PrixActions.class, contexte );
   }
}
```

Si une instance de `AppAppelCompteur` est toujours en cours d'exécution, arrêtez-la car elle utilise les mêmes adresses et ports que cette nouvelle application. Une fois `AppPrixActions3` démarrée, l'espace de nom utilisé pour qualifier le nom de méthode dans les messages SOAP sera `urn:ServicePrixActions`. Voyons la réponse générée par le serveur à cette requête :

```
<soap:Envelope
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
 soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
   <soap:Body>
     <n:getPrixReponse xmlns:n='urn:ServicePrixActions'>
       <Result xsi:type='xsd:float'>75.33</Result>
     </n:getPrixReponse>
   </soap:Body>
 </soap:Envelope>
```

L'enveloppe de la réponse est plus ou moins la même que celle générée par Apache SOAP. La seule vraie différence est que la valeur de retour est contenue dans un élément nommé `Result` alors que sous Apache SOAP, cet élément est nommé `return`. Mais c'est la structure de la réponse qui nous importe et vous pouvez voir qu'elle est la même que celle du serveur Apache SOAP.

Utilisation des paramètres in, out et in/out

Dans tous les exemples vus jusqu'à présent, nous avons supposé que les paramètres de méthodes contenaient des valeurs passées du client vers le service et que ce service ne modifiait pas ces valeurs avant de répondre. Ceci est le comportement par défaut des

services RPC SOAP. Les paramètres sont dans ce cas considérés comme des paramètres `in` car leur valeur est passée « vers l'intérieur » du service. Mais SOAP supporte également l'usage de deux autres styles de paramètres. Les méthodes de services peuvent comporter des paramètres ne contenant aucune donnée envoyée par le client mais envoyée à celui-ci lors du retour de la méthode. Il s'agit des paramètres `out` : la valeur du paramètre est passée « vers l'extérieur » du service. Les paramètres peuvent également être `in/out`, lorsqu'une valeur est passée au service et qu'une autre valeur est renvoyée.

Toutes les implémentations SOAP ne supportent pas l'usage des paramètres `out` ou `in/out` (Apache SOAP ne le fait pas, GLUE oui). Ceci devrait en soi vous alarmer. Mieux vaut ne pas utiliser une fonctionnalité qui n'est pas supportée partout, à moins que vous ne contrôliez aussi bien les serveurs que leurs clients. Vous devriez pouvoir vous passer des paramètres `out` ou `in/out` en définissant un type personnalisé contenant les valeurs que vous voulez renvoyer puis en définissant votre méthode de service pour qu'elle renvoie une instance de ce type personnalisé. Ceci est mon approche. Elle évite de plus toute confusion sur le but de la méthode. D'un autre côté, les implémentations les plus récentes supportent les paramètres `in/out`. En fait, la plupart utilisent la technique que nous allons examiner.

Maintenant que je vous ai mis en garde sur leur usage, voyons comment les paramètres `out` et `in/out` peuvent être utilisés. GLUE utilise des classes porteuses qui encapsulent les types de données lors de l'utilisation de paramètres `out` et `in/out`. Il existe déjà des classes porteuses pour les primitives Java et les classes courantes comme `java.lang.String`. Pour les types de données personnalisés, GLUE comporte un utilitaire `holder` qui génère les classes porteuses à votre place. (Gardez à l'esprit que cette approche est spécifique à GLUE. SOAP ne dicte pas l'API mais seulement l'encodage XML.) Voici une version modifiée de la classe `PrixActions` comportant deux nouvelles méthodes. La première est `getPrixOut()`. Elle renvoie une valeur `booléenne` indiquant si la requête a été couronnée de succès. Les deux premiers paramètres sont les mêmes que ceux de la méthode `getPrix()` originale : un symbole d'action et la devise. Le troisième paramètre est nommé `prix`. Il s'agit d'un paramètre `out` n'ayant pas de valeur lors de l'invocation de la méthode et contenant le prix envoyé en résultat lors du retour de la méthode. Ainsi `getprix()` utilise un paramètre `out` pour renvoyer le prix calculé par la méthode `getPrix()` d'origine. Le paramètre `prix` est une instance de `electric.util.holder.floatOut` qui est une classe porteuse pour les paramètres `out` en virgule flottante.

L'autre méthode nouvelle est nommée `corrigeSymbole()` et convertit un symbole d'action au format approprié (caractères majuscules). Elle renvoie un `booléen` indiquant le succès ou l'échec mais plus important, elle utilise un paramètre `in/out String` nommé `symbole`. Ce paramètre contient un symbole d'action lorsque la méthode est appelée. Au retour de `corrigeSymbole()` il contient le symbole corrigé de l'action. Pour permettre ce comportement, le paramètre de symbole est déclaré comme une instance de `electric.util.holder.java.lang.StringInOut`, une classe porteuse pour les paramètres `String in/out`. `AppPrixActions3` peut être utilisée pour publier le service sans modifications.

```
package javasoap.livre.ch4;
import electric.util.holder.*;
import electric.util.holder.java.lang.*;
public class PrixActions {
```

```
    public float getPrix(String action, String devise) {
       float resultat;
       // détermine le prix de l'action et le renvoie
       // dans la devise spécifiée
       resultat = (float)75.33;
       return resultat;
    }
    public boolean getPrixOut(String action, String devise,
                        floatOut prix) {
       try {
          prix.value = getPrix(action, devise);
          return true;
       }
       catch (Exception e) {
          return false;
       }
    }
    public boolean corrigeSymbole(StringInOut symbole) {
       symbole.value = symbole.value.toUpperCase();
       return true;
    }
}
```

Dans la mesure où nous avons modifié le service en y ajoutant des méthodes, nous allons devoir modifier l'interface `javasoap.livre.ch4.IServicePrixActions` en conséquence. Voici la nouvelle interface :

```
package javasoap.livre.ch4;
import electric.util.holder.*;
import electric.util.holder.java.lang.*;
public interface IServicePrixActions {
  float getPrix(String action, String devise);
  public boolean getPrixOut(String action, String devise,
                      floatOut prix);
  public boolean corrigeSymbole(StringInOut symbole);
}
```

Créons maintenant l'application qui utilisera ce service. Nous appellerons d'abord `corrigeSymbole()` pour convertir le symbole de l'action en majuscules en utilisant un paramètre `in/out`, puis appellerons `getPrixOut()` pour récupérer le prix à l'aide d'un paramètre `out` :

```
package javasoap.livre.ch4;
import electric.registry.Registry;
import electric.registry.RegistryException;
import electric.util.holder.*;
import electric.util.holder.java.lang.*;
public class AppInOut {
   public static void main(String[] args) throws Exception
   {
      try {
         IServicePrixActions ctr =
```

```
            (IServicePrixActions)Registry.bind(
              "http://localhost:8004/glue/urn:ServicePrixActions.wsdl",
              IServicePrixActions.class);
          StringInOut action = new StringInOut("MiNdStRm");
          boolean ok = ctr.corrigeSymbole(action);
          if (ok) {
            System.out.println("Symbole converti en : "
                        + action.value);
          }
          String devise = "USD";
          floatOut prix = new floatOut();
          ok = ctr.getPrixOut(action.value, devise, prix);
          System.out.println("Le résultat est " + prix.value);
        }
        catch (RegistryException e)
        {
          System.out.println(e);
        }
    }
}
```

La variable du symbole de l'action, `action`, est instanciée avec le symbole `"MiNdStRm"`. Nous passons alors cette variable, qui est une instance de `electric.util.holder.java.lang.StringInOut` à la méthode `corrigeSymbole()` de l'interface du service. Au retour de l'appel, nous attendons que le symbole ait été converti en majuscules. Nous appelons ensuite `getPrixOut()` en lui passant `action.value` comme premier paramètre. La devise est passée comme auparavant et une instance de `electric.holder.floatOut` nommée `prix` est passée en dernier paramètre. Au retour de la méthode, `prix` contient le prix de l'action. L'exécution de l'exemple produit ce résultat :

```
Symbole converti en : MINDSTRM
Le résultat est 75.33
```

Comme vous pouvez le voir, les paramètres `out` et `in/out` ont été gérés comme prévu. Avant de terminer, voyons les parties des enveloppes SOAP échangées entre le client et le serveur qui ont été affectées par notre usage des paramètres `out` et `in/out`. L'appel à `corrigeSymbole()` produit en réponse une enveloppe légèrement différente de celles que nous avons vu jusqu'à maintenant. La valeur booléenne renvoyée est immédiatement suivie de la valeur du paramètre modifiée. Voici le corp SOAP de cette réponse :

```
<soap:Body>
  <n:corrigeSymboleResponse xmlns:n='urn:ServicePrixActions'>
    <Result xsi:type='xsd:boolean'>true</Result>
    <arg0 xsi:type='xsd:string'>MINDSTRM</arg0>
  </n:corrigeSymboleResponse>
</soap:Body>
```

La valeur modifiée du paramètre `in/out` de l'élément XML `arg0` apparaît juste après l'élément `Result`. Et naturellement, vous pouvez voir que la valeur d'origine a été convertie en majuscules. Voyons maintenant la partie de l'enveloppe envoyée au serveur lorsque `getPrixOut()` a été invoquée. Il n'y a que deux valeurs de paramètres encodées car le troisième est de type `out` et n'a donc pas de valeur au moment de l'invocation de la méthode.

```
<n:getPrixOut xmlns:n='urn:ServicePrixActions'>
  <arg0 xsi:type='xsd:string'>MINDSTRM</arg0>
  <arg1 xsi:type='xsd:string'>USD</arg1>
</n:getPrixOut>
```

Pour terminer, voici la partie du corps SOAP correspondant à la réponse renvoyée lors de l'appel à `getPrixOut()`. La valeur du paramètre `prix` suit le résultat lui-même. Les deux premiers paramètres n'ont pas de valeur car ils sont tous deux des paramètres `in` et voient donc leurs valeurs inchangées.

```
<n:getPrixOutResponse xmlns:n='urn:ServicePrixActions'>
  <Result xsi:type='xsd:boolean'>true</Result>
  <arg2 xsi:type='xsd:float'>75.33</arg2>
</n:getPrixOutResponse>
```

Au chapitre suivant, nous commencerons à nous intéresser à des fonctions et des possibilités plus complexes de SOAP et de ses implémentations.

5

Utilisation des types de données complexes

Dans le chapitre précédent, nous avons créé des services de type RPC en Java. Ces services n'avaient à manipuler que des types de données simples. Ces derniers suffisent dans la plupart des situations, mais vous pourriez avoir besoin d'utiliser des types plus complexes tels que ceux que vous utilisez dans votre travail quotidien de programmation Java. Nous verrons dans ce chapitre comment créer des services avec des paramètres de méthodes ou des valeurs de retour de types tableaux ou JavaBeans. Nous n'aborderons pas les types de données personnalisés avant le chapitre suivant, dans la mesure où il est possible que tous les types personnalisés que vous créerez utiliseront les types complexes que nous allons traiter maintenant.

Passage de tableaux en paramètres

Soyons clair : les tableaux sont probablement le type de données complexe le plus courant en programmation. On les retrouve partout et leur usage sous SOAP est donc crucial. Nous avons traité les tableaux SOAP dans le détail au chapitre 3 et vous devez désormais être au fait de la façon dont ils sont encodés. Passons donc directement à l'écriture d'un programme Java les utilisant.

Nous avons commencé à utiliser des exemples liés au marché boursier et allons donc continuer dans cette voie. Il pourrait être utile d'avoir un service renvoyant des informations sur un ensemble de symboles boursiers. Il fournirait le volume total d'actions échangées dans la journée, le prix d'échange moyen de ces actions, le plus haut volume d'échange de la journée etc. Il existe de nombreuses possibilités. Commençons par un service renvoyant le nombre total d'actions échangées durant la journée. Le service s'appelle `urn:ServiceBoursierDeBase` et contient une méthode nommée `getVolumeTotal`. Voici la classe Java qui implémente le service :

```
package javasoap.livre.ch5;
public class ServiceBoursierDeBase {

   public ServiceBoursierDeBase() {}

   public int getVolumeTotal(String[] actions) {
```

```
        // récupère le volume de chaque action à partir
        // d'une source de données quelconque et renvoie le total
        int total = 345000;
        return total;
    }
}
```

La classe ServiceBoursierDeBase contient la méthode getVolumeTotal(), qui renvoie le nombre total d'actions échangées. Comme nous n'allons accéder à aucune source de données, nous renverrons une valeur constante. La méthode renvoie un entier et reçoit un seul paramètre nommé actions qui est un tableau de valeurs de type String. Les chaînes du tableau sont les symboles des actions. Dans une application réelle, nous récupérerions le volume de chaque action dans une source de données et en renverrions la somme.

Apache SOAP supporte nativement les tableaux et vous n'aurez donc rien de particulier à faire du côté du service. Cela signifie également qu'il n'y aura rien de nouveau dans le descripteur de déploiement. Il est élaboré de la même façon qu'il l'était dans le chapitre précédent. Voici le descripteur de déploiement du service urn:ServiceBoursierDeBase:

```
<isd:service
      xmlns:isd="http://xml.apache.org/xml-soap/deployment"
      id="urn:ServiceBoursierDeBase">
  <isd:provider type="java"
      scope="Application"
      methods="getVolumeTotal">
    <isd:java class="javasoap.livre.ch5.ServiceBoursierDeBase"
      static="false"/>
  </isd:provider>
  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
  </isd:mappings>
</isd:service>
```

Vous pouvez déployer le service à l'aide de ce descripteur ou de l'outil d'administration Apache SOAP depuis votre navigateur.

Écrivons maintenant l'application cliente qui accèdera au service. Elle est similaire à certains des exemples du chapitre précédent. Elle diffère seulement dans le fait que le paramètre que nous passons est un tableau d'instances de String et non un objet simple. En voici le code:

```
package javasoap.livre.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class ClientVolume {
   public static void main(String[] args) throws Exception {

      URL url = new URL("http://localhost:8080/soap/servlet/rpcrouter");
```

```
        Call appel = new Call();
        appel.setTargetObjectURI("urn:ServiceBoursierDeBase");

        appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
        String[] actions = { "MINDSTRM", "MSFT", "SUN" };
        Vector params = new Vector();
        params.addElement(new Parameter("actions",
                          String[].class, actions, null));
        appel.setParams(params);

        try {
           appel.setMethodName("getVolumeTotal");
           Response rep = appel.invoke(url, "");
           Parameter ret = rep.getReturnValue();
           Object valeur = ret.getValue();
           System.out.println("Le volume total est de " + valeur);
        }
        catch (SOAPException e) {
           System.err.println("SOAPException (" +
                         e.getFaultCode() + "): " +
                         e.getMessage());
        }
     }
  }
```

Nous avons passé `String[].class` en second paramètre du constructeur `Parameter`. Ceci identifie la variable `actions` comme un tableau de chaînes. Nous n'avons besoin de rien d'autre. Si vous exécutez cet exemple, le résultat devrait être :

```
Le volume total est de 345000
```

Examinons l'enveloppe SOAP envoyée au serveur pour invoquer la méthode `getVolumeTotal` du service :

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <SOAP-ENV:Body>
    <ns1:getVolumeTotal xmlns:ns1="urn:ServiceBoursierDeBase"
      SOAP-ENV:encodingStyle=
           "http://schemas.xmlsoap.org/soap/encoding/">
      <actions>
        xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
        xsi:type="ns2:Array"
        ns2:arrayType="xsd:string[3]">
           <item xsi:type="xsd:string">MINDSTRM</item>
           <item xsi:type="xsd:string">MSFT</item>
           <item xsi:type="xsd:string">SUN</item>
      </actions>
```

```
    </ns1:getVolumeTotal>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

L'élément actions représente le tableau de chaînes que nous passons en paramètre à la méthode du service. Il est typé en tableau par l'affectation à l'attribut xsi:type de la valeur ns2:Array. L'identifiant d'espace de noms ns2 a été défini dans la ligne qui précède. Puis l'attribut ns2:arrayType s'est vu assigner la valeur xsd:string[3]. Cela signifie qu'il est un tableau de taille 3 et que chacun de ses éléments est un xsd:string. On trouve trois éléments enfants d'actions, chacun d'eux nommé item. Rappelez-vous que le nom utilisé pour les éléments du tableau n'a pas d'importance et que les différentes implémentations SOAP utilisent des systèmes différents pour nommer ces éléments. Chaque élément est explicitement typé en positionnant l'attribut xsi:type à la valeur xsd:string.

Cet exemple utilise un tableau homogène, ce qui signifie que tous les éléments du tableau sont des instances du même type. Vous pourrez dans certaines occasions utiliser également des tableaux hétérogènes et nous allons donc examiner cette possibilité. En Java, les tableaux sont souvent utilisés comme paramètres de méthodes qui, dans d'autres langages, recevraient une liste de paramètres de longueur variable. La fonction printf() du langage C par exemple, ne possède pas un nombre fixe de paramètres. Bien que Java ne possède pas cette capacité, il est possible de la simuler en passant des valeurs de paramètres à l'intérieur d'un tableau. Les tableaux Java comportent des objets du même type, mais ce type peut être Object, ce qui autorise tout objet (un tableau de type Object[] peut contenir tout objet Java dans la mesure où ils dérivent tous de la classe Object).

Ajoutons à urn:ServiceBoursierDeBase une méthode qui recevra un tableau hétérogène en paramètre. La méthode se nomme executeOrdre. Son paramètre est un tableau contenant un symbole d'action, le nombre d'actions à échanger et un drapeau indiquant s'il s'agit d'un ordre de vente ou d'achat (true signifiant achat). La valeur de retour est une chaîne de caractères décrivant l'ordre.[1] Voici la classe ServiceBoursierDeBase modifiée :

```
package javasoap.livre.ch5;
public class ServiceBoursierDeBase {

   public ServiceBoursierDeBase() {
   }
   public int getVolumeTotal(String[] actions) {

      // récupère le volume de chaque action à partir
      // d'une source de données quelconque et get le total

      int total = 345000;
      return total;
```

1. Il existe certainement d'autres façons de concevoir une interface vers une méthode comme celle-ci et ce n'est pas la façon que je choisirait. Cette situation appelle probablement l'utilisation d'une classe personnalisée ou d'un bean Java. L'approche que j'utilise dans cet exemple montre toutefois l'utilisation de tableaux hétérogènes.

```
    }
    public String executeOrdre(Object[] params) {
       String resultat;
       try {
          String action = (String)params[0];
          Integer nombreActions = (Integer)params[1];
          Boolean achat = (Boolean)params[2];
          String typeOrdre = "Achat";
          if (false == achat.booleanValue()) {
             typeOrdre = "Vente";
          }
          resultat = (typeOrdre + " de " + nombreActions + " actions " + action);
       }
       catch (ClassCastException e) {
          resultat = "Erreur dans le type de paramètre ";
       }
       return resultat;
    }
}
```

Il n'y a qu'un paramètre dans la méthode executeOrdre(), un Object[] nommé params. Les objets de ce tableau doivent être transtypés dans leurs types correspondants: String Integer et Boolean. J'ai l'habitude de mettre les transtypages de classes à l'intérieur d'un bloc try/catch pour prévenir les éventuelles erreurs lors de l'appel. Je conserve ainsi un moyen d'action si la méthode est appelée de façon incorrecte, même si cela ne consiste qu'à renvoyer une description de l'erreur. Au chapitre 7, nous verrons les fautes SOAP destinées à ce genre de situations. L'information transmise dans le tableau est utilisée pour générer une chaîne décrivant les paramètres, cette chaîne étant stockée dans la variable resultat renvoyée à l'appelant.

Nous pouvons maintenant modifier l'application cliente pour qu'elle envoie un Object[] approprié en paramètre à la méthode executeOrdre. La variable multiParams est déclarée comme un Object[] et est renseignée par la String MINDSTRM, un Integer valant 100 et un Boolean de valeur true. Dans la mesure où nous utilisons un tableau d'instances d'Object Java, nous n'utilisons pas de primitives Java comme éléments du tableau. Au lieu de cela, nous encapsulons les valeurs dans des objets Java correspondants. Le second paramètre du constructeur Parameter est un Object[].class représentant la classe d'un tableau d'instances d'objets.

```
package javasoap.livre.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class ClientBoursier {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");
```

```
        Call appel = new Call();
        appel.setTargetObjectURI("urn:ServiceBoursierDeBase");

        appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
        Object[] multiParams = { "MINDSTRM", new Integer(100),
                                 new Boolean(true) };
        Vector params = new Vector();
        params.addElement(new Parameter("params",
                          Object[].class, multiParams, null));
        appel.setParams(params);

        try {
           appel.setMethodName("executeOrdre");
           Response rep = appel.invoke(url, "");
           Parameter ret = rep.getReturnValue();
           Object valeur = ret.getValue();
           System.out.println("Description de l'ordre : " + valeur);
        }
        catch (SOAPException e) {
           System.err.println("SOAPException (" +
                         e.getFaultCode() + "): " +
                         e.getMessage());
        }
     }
  }
```

Si tout se déroule bien, le résultat de l'exécution de la méthode de service executeOrdre() sera :

```
Description de l'ordre : Achat de 100 MINDSTRM
```

Nous pouvons provoquer un échec de l'objet du service en changeant l'ordre des paramètres dans le tableau multiParams. Nous obtiendrons dans ce cas une erreur de transtypage de classe et la méthode renverra une erreur.

Voici l'enveloppe SOAP appropriée à l'invocation de la méthode de service executeOrdre() :

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <SOAP-ENV:Body>
    <ns1:executeOrdre xmlns:ns1="urn:ServiceBoursierDeBase"
      SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">

      <params
          xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
          xsi:type="ns2:Array" ns2:arrayType="xsd:anyType[3]">
        <item xsi:type="xsd:string">MINDSTRM</item>
```

```
        <item xsi:type="xsd:int">100</item>
        <item xsi:type="xsd:boolean">true</item>
      </params>
    </ns1:executeOrdre>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

L'élément `params` est typé par affectation à l'attribut `xsi:type` de la valeur `ns2:Array`. La seule différence avec l'exemple homogène est que chaque élément du tableau possède une valeur différente assignée à l'attribut `ns2:arrayType`. La valeur `xsd:anyType[3]` indique que le tableau contient trois éléments, chacun d'eux pouvant être d'un type quelconque de données valide.[2]

Voyons maintenant comment passer des tableaux en paramètres sous GLUE. La classe `ServiceBoursierDeBase` peut être déployée sous GLUE sans modification. Nous utiliserons une application Java simple pour démarrer ce service :

```
package javasoap.livre.ch5;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
public class AppBoursiereDeBase {
   public static void main( String[] args )
      throws Exception {

      HTTP.startup("http://localhost:8004/glue");
      Context contexte = new Context();
      contexte.addProperty("activation", "application");
      contexte.adcProperty("namespace",
                           "urn:ServiceBoursierDeBase");
      Registry.publish("urn:ServiceBoursierDeBase",
         javasoap.livre.ch5.ServiceBoursierDeBase.class, contexte );
   }
}
```

Compiler et exécuter l'application déploiera le service. Écrivons maintenant un petit exemple simple pour accéder au service à l'aide de l'API GLUE. Examinons d'abord l'interface du service, `IServiceBoursierDeBase` :

```
package javasoap.livre.ch5;
public interface IServiceBoursierDeBase {
  int getVolumeTotal(String[] symboles);
  String executeOrdre(Object[] params);
}
```

Nous pouvons maintenant écrire une application qui s'attache au service et appelle ses méthodes :

---

2. Au chapitre 3, nous avons évoqué la possibilité d'utiliser `ur-type` pour représenter tout type de données possible. Nous voyons ici l'utilisation de anyType pour cet usage. La recommandation XML Schema Part 0, datée du 2 mai 2001, explique en ces termes dans la section 2.5.4 : « anyType représente une abstraction appelée l'ur-type qui est le type de base duquel tous les types simples et complexes sont dérivés. Un anyType ne fait porter aucune contrainte sur son contenu. »

```
package javasoap.livre.ch5;
import electric.registry.RegistryException;
import electric.registry.Registry;
public class ClientBoursierDeBase {
   public static void main(String[] args) throws Exception
   {
      try {
        IServiceBoursierDeBase srv =
          (IServiceBoursierDeBase)Registry.bind(
            "http://localhost:8004/glue/urn:ServiceBoursierDeBase.wsdl",
            IServiceBoursierDeBase.class);
        String[] actions = { "MINDSTRM", "MSFT", "SUN" };
        int total = srv.getVolumeTotal(actions);
        System.out.println("Le volume total est de " + total);
        Object[] multiParams = { "MINDSTRM", new Integer(100),
                                 new Boolean(true) };
        String desc = srv.executeOrdre(multiParams);
        System.out.println("Description de l'ordre : " + desc);
     }
     catch (RegistryException e)
     {
        System.out.println(e);
     }
   }
}
```

Comme nous l'avons vu auparavant, GLUE nous permet d'utiliser une syntaxe de programmation Java familière sans nous préoccuper des constructions SOAP sous-jacentes. Cela reste vrai pour le passage de paramètres sous forme de tableaux. Tout est pris en main pour nous, une fois l'interface liée au service.

Il y a un certain nombre de choses intéressantes à voir dans l'enveloppe de la requête SOAP générée par cet exemple. Voici l'enveloppe SOAP destinée à l'invocation de la méthode `getVolumeTotal()` :

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:getVolumeTotal xmlns:n='urn:ServiceBoursierDeBase'>
        <arg0 href='#id0'/>
      </n:getVolumeTotal>
      <id0 id='id0' soapenc:root='0'
        xmlns:ns2='http://www.themindelectric.com/package/java.lang/'
        xsi:type='soapenc:Array' soapenc:arrayType='xsd:string[3]'>
         <i xsi:type='xsd:string'>MINDSTRM</i>
         <i xsi:type='xsd:string'>MSFT</i>
         <i xsi:type='xsd:string'>SUN</i>
      </id0>
```

```
    </soap:Body>
  </soap:Envelope>
```

L'élément Body vient en premier comme nous l'avons déjà vu. L'enveloppe est qualifiée par l'identifiant soap, qui représente l'espace de noms de l'enveloppe SOAP. Le premier élément enfant du corps SOAP est getVolumeTotal, qui est naturellement le nom de la méthode de service invoquée. getVolumeTotal est qualifiée par un espace de noms à l'aide du nom du service. Le seul élément enfant de getVolumeTotal est arg0, qui représente le paramètre passé à la méthode. Nous ne passons pas en réalité le tableau mais une référence à celui-ci. Ceci constitue une différence majeure dans la façon dont GLUE et l'API Apache SOAP génèrent cet appel. Apache SOAP place le tableau dans l'enveloppe en tant qu'élément enfant de getVolumeTotal alors que GLUE utilise une référence et sérialise le tableau une fois l'élément getVolumeTotal terminé. Le paramètre est donc sérialisé en tant que arg0 et inclut un attribut href possédant la valeur #id0. Aucune donnée n'est incluse, le tableau se trouvant en réalité ailleurs.

Le tableau que nous avons passé en paramètre suit l'élément getVolumeTotal. Il est nommé id0, bien que le nom de l'élément lui-même, généré par GLUE, ne soit pas important. L'attribut id reçoit la valeur id0, qui correspond à la valeur href utilisée dans l'élément getVolumeTotal. GLUE génère l'attribut soapenc:root en lui affectant la valeur 0, signifiant par là que cet élément n'est pas considéré comme la racine d'un graphe d'objet. (GLUE semble inclure cet attribut automatiquement.) Apparaît ensuite la déclaration d'un identifiant d'espace de noms nommé ns2 qui semble identifier le package GLUE interne pour java.lang. L'identifiant ns2 n'est toutefois jamais utilisé. Les attributs xsi:type et soapenc:arrayType sont positionnés de la même façon que dans les exemples Apache SOAP. Pour finir, les éléments du tableau sont sérialisés. La seule différence entre cet exemple et celui généré par Apache SOAP réside dans le nom des éléments du tableau. Apache SOAP nomme ces éléments item, et GLUE i. Les noms importent peu et le résultat est le même.

Cet exemple nous donne une bonne occasion de voir deux façons l'une comme l'autre valide, de sérialiser les tableaux. Il est important que les implémentations SOAP comprennent ces différents styles si elles doivent interopérer. C'est l'une des raisons pour lesquelles nous continuons à montrer les enveloppes générées par les exemples. Vous familiariser avec les différents styles de sérialisation vous aidera si vous rencontrez des problèmes à faire communiquer correctement des applications basées sur des implémentations différentes. (Ai-je dit *si ?*)

Voyons à quoi ressemble l'enveloppe SOAP pour l'appel de la méthode de service executeOrdre. Cette méthode reçoit en paramètre un tableau hétérogène. Elle utilise également une référence à un tableau sérialisé séparément, encodé cette fois comme xsd:anyType :

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:executeOrdre xmlns:n='urn:ServiceBoursierDeBase'>
```

```
        <arg0 href='#id0'/>
      </n:executeOrdre>
      <id0 id='id0' soapenc:root='0'
        xmlns:ns2='http://www.themindelectric.com/package/java.lang/'
        xsi:type='soapenc:Array' soapenc:arrayType='xsd:anyType[3]'>
        <i xsi:type='xsd:string'>MINDSTRM</i>
        <i xsi:type='xsd:int'>100</i>
        <i xsi:type='xsd:boolean'>true</i>
      </id0>
    </soap:Body>
</soap:Envelope>
```

Renvoyer des tableaux

Jusqu'à présent, nous avons passé les tableaux en paramètres. Utilisons-les maintenant comme valeur de retour d'une méthode de service. Nous allons ajouter une méthode à notre service, nommée `getLaPlusActive()`, qui renverra une `String[]` contenant les symboles des actions les plus échangées de la journée. Voici la nouvelle version de la classe `ServiceBoursierDeBase` :

```
package javasoap.livre.ch5;
public class ServiceBoursierDeBase {

   public ServiceBoursierDeBase() {
   }
   public String[] getLaPlusActive() {

      // renvoie les actions les plus échangées
      String[] actives = { "ABC", "DEF", "GHI", "JKL" };
      return actives;
   }
   public int getVolumeTotal(String[] actions) {

      // récupère le volume de chaque action
      // depuis une source de données et renvoie le total
      int total = 345000;
      return total;
   }
   public String executeOrdre(Object[] params) {
      String resultat;
      try {
         String action = (String)params[0];
         Integer nbParts = (Integer)params[1];
         Boolean achat = (Boolean)params[2];
         String typeOrdre = "Achat";
         if (false == achat.booleanValue()) {
            typeOrdre = "Vente";
         }
         resultat = (typeOrdre + " de " + nbParts + " " + action);
      }
```

```
        catch (ClassCastException e) {
           resultat = "Type de paramètre incorrect ";
        }
        return resultat;
     }
  }
```

Dans la mesure où nous n'accédons pas réellement à une source de données, nous avons placé des symboles d'actions fictives dans un tableau que nous renvoyons. Déployons maintenant le service. Appeler cette méthode de service depuis un client Apache SOAP est très simple. Il n'y a pas de paramètres à envoyer à la méthode et nous avons juste à préparer et à effectuer l'appel :

```
package javasoap.livre.ch5;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class ClientPlusActive
{
  public static void main(String[] args) throws Exception
  {
    URL url = new
      URL("http://localhost:8080/soap/servlet/rpcrouter");
    Call appel = new Call();
    appel.setTargetObjectURI("urn:ServiceBoursierDeBase");
    appel.setMethodName("getLaPlusActive");
    Response rep;
    try {
      rep = appel.invoke(url, "");
      Parameter ret = rep.getReturnValue();
      String[] valeur = (String[])ret.getValue();
      int cnt = valeur.length;
      for (int i = 0; i < cnt; i++) {
         System.out.println(valeur[i]);
      }
    }
    catch (SOAPException e) {
      System.err.println("SOAPException (" +
                         e.getFaultCode() + "): " +
                         e.getMessage());
    }
  }
}
```

Nous avons transtypé la valeur de retour de `ret.getValue` en `String[]`, dans la mesure où c'est le type que nous attendons en retour. Dans les exemples précédents, nous avons pu conserver la valeur en instance d'`Object` car nous avons confié à la méthode d'objet `toString()` le soin d'afficher la valeur. Dans le cas présent, nous avons besoin d'itérer à travers le tableau et il est donc nécessaire de transtyper la valeur en un type de tableau approprié. Après cela, nous n'aurons plus qu'à déterminer la longueur du tableau, à

boucler dans ses valeurs et à les afficher au fur et à mesure. En exécutant cet exemple, vous devez obtenir le résultat suivant :

```
ABC
DEF
GHI
JKL
```

L'enveloppe SOAP renvoyée lors de l'invocation de cette méthode est assez simple :

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:getLaPlusActiveReponse
        xmlns:ns1="urn:ServiceBoursierDeBase"
        SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">

      <return
        xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
        xsi:type="ns2:Array" ns2:arrayType="xsd:string[4]">
          <item xsi:type="xsd:string">ABC</item>
          <item xsi:type="xsd:string">DEF</item>
          <item xsi:type="xsd:string">GHI</item>
          <item xsi:type="xsd:string">JKL</item>
      </return>
     </ns1:getLaPlusActiveReponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Pour déployer cette version de la classe ServiceBoursierDeBase sous GLUE, nous pouvons utiliser notre bonne vieille classe AppBoursiereDeBase. Nous pouvons modifier l'interface Java IServiceBoursierDeBase, pour y inclure la nouvelle méthode :

```
package javasoap.livre.ch5;
public interface IServiceBoursierDeBase {
  int getVolumeTotal(String[] symboles);
  String executeOrdre(Object[] params);
  String[] getLaPlusActive();
}
```

Modifions maintenant l'application ClientBoursierDeBase pour y inclure un appel à la méthode getLaPlusActive(), puis itérons au travers des valeurs du tableau et affichons le résultat. Sous GLUE, nous n'avons pas à transtyper la valeur de retour en String[] car contrairement à l'exemple Apache SOAP, la méthode getLaPlusActive() de l'interface est définie pour renvoyer le type approprié. Voici le code modifié :

```
package javasoap.livre.ch5;
import electric.registry.RegistryException;
import electric.registry.Registry;
public class ClientBoursierDeBase {
   public static void main(String[] args) throws Exception
```

```
    {
       try {
         IServiceBoursierDeBase srv = (IServiceBoursierDeBase)Registry.bind(
           "http://localhost:8004/glue/urn:ServiceBoursierDeBase.wsdl",
           IServiceBoursierDeBase.class);
         String[] actions = { "MINDSTRM", "MSFT", "SUN" };
         int total = srv.getVolumeTotal(actions);
         System.out.println("Le volume total est de " + total);
         Object[] multiParams = { "MINDSTRM", new Integer(100),
                                  new Boolean(true) };
         String desc = srv.executeOrdre(multiParams);
         System.out.println("Description de l'ordre : " + desc);
         String[] actives = srv.getLaPlusActive();
         int cnt = actives.length;
         for (int i = 0; i < cnt; i++) {
            System.out.println(actives[i]);
         }
      }
      catch (RegistryException e)
      {
         System.out.println(e);
      }
    }
}
```

En exécutant cet exemple, vous obtiendrez le résultat suivant:

```
Le volume total est de 345000
Description de l'ordre : Achat de 100 actions MINDSTRM
ABC
DEF
GHI
JKL
```

GLUE use de la même technique de sérialisation pour les tableaux employés comme valeurs de retour que pour les tableaux employés comme paramètres. Il utilise une référence à un tableau sérialisé séparément comme valeur de retour, et référence le tableau de données réelles. L'enveloppe SOAP renvoyée à l'invocation de la méthode `getLaPlusActive()` est:

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:renvoiLaPlusActiveReponse
        xmlns:n='urn:ServiceBoursierDeBase'>
         <Result href='#id0'/>
      </n:renvoiLaPlusActiveReponse>
      <id0 id='id0' soapenc:root='0'
```

```
        xmlns:ns2='http://www.themindelectric.com/package/java.lang/'
        xsi:type='soapenc:Array' soapenc:arrayType='xsd:string[4]'>
         <i xsi:type='xsd:string'>ABC</i>
         <i xsi:type='xsd:string'>DEF</i>
         <i xsi:type='xsd:string'>GHI</i>
         <i xsi:type='xsd:string'>JKL</i>
      </id0>
    </soap:Body>
  </soap:Envelope>
```

Passage de types personnalisés en paramètres

En tant que programmeurs Java, nous n'allons certainement pas nous limiter aux classes fournies par les packages Java standard. Une part substantielle de notre code est constituée de types personnalisés et de classes Java de notre cru, qui incarnent les fonctionnalités et caractéristiques des systèmes que nous élaborons.

Considérons la conception d'une classe Java contenant toutes les données nécessaires pour spécifier un échange boursier. Cette nouvelle classe pourra contenir le symbole de l'action échangée, le nombre de parts à échanger et une indication sur le type d'ordre (achat ou vente). Lors de la conception d'une telle classe, il est important de la situer dans le contexte d'un système plus large. Ce type d'analyse produit des indices qui peuvent gouverner les décisions concernant les propriétés et comportements à donner à la classe. Nous faisons tous cela et tout le temps. Cela s'appelle la conception de logiciel. Le résultat est une classe Java contenant des méthodes accédant à des propriétés, ainsi qu'un comportement donné. SOAP consistant à transporter des données, nous allons nous intéresser aux propriétés de la classe. C'est cela que nous voudrons acheminer au travers du réseau.

Une façon commune d'exprimer les propriétés d'une classe Java consiste à utiliser les modèles de conception de JavaBeans. Ces modèles spécifient une convention de nommage à utiliser pour les méthodes d'accès à une classe. Vous n'êtes peut-être pas un familier des JavaBeans, mais je suis prêt à parier que vous avez rencontré ces modèles de nombreuses fois. Voici comment le modèle d'accès à une propriété est décrit dans le livre *Developing Java Beans*, paru chez O'Reilly & Associates :

> Les méthodes utilisées pour récupérer ou positionner les valeurs de propriétés doivent se conformer au standard des modèles de conception des propriétés. Ces méthodes sont autorisées à renvoyer des exceptions vérifiées si besoin est, mais cela est optionnel. Les signatures de méthodes se présentent comme suit :
>
> ```
> public void set<NomPropriété>(<TypePropriété> valeur);
> public <TypePropriété> get<NomPropriété>();
> ```
>
> L'existence d'une paire de méthodes appariées, se conformant à ce modèle représente une propriété en lecture/écriture de nom <NomPropriété> et de type <TypePropriété>. Si seule la méthode get existe, la propriété est considérée en lecture seulement et si seule la méthode set existe, elle est considérée en écriture seulement. Dans le cas où le <TypePropriété> est booléen, la méthode get peut être remplacée ou complétée par une méthode utilisant la signature suivante :
>
> ```
> public boolean is<NomPropriété>();
> ```

En suivant ce modèle pour le nommage des accesseurs de propriétés, vous permettez au méthodes de l'accesseur d'être déterminées lors de l'exécution grâce au mécanisme d'introspection Java.[3] C'est une manière commode, offerte aux implémentations SOAP, d'accéder aux valeurs d'une instance de classe Java pour sérialiser les données d'un message SOAP. Il se trouve qu'aussi bien Apache SOAP que GLUE tirent parti de la réflexion. Cela signifie que vous aurez juste à respecter une convention de nommage bien établie pour être en mesure d'utiliser des classes personnalisées sous SOAP. Il y aura naturellement un petit travail supplémentaire auquel nous allons nous atteler maintenant.

Définissons tout d'abord un ordre en bourse en termes de données. Il doit comporter un symbole représentant l'action échangée. Ce symbole doit être une chaîne de caractères. Il nécessite également un indicateur booléen spécifiant s'il s'agit d'un ordre d'achat ou de vente. Enfin, il nécessite un entier indiquant le nombre de parts à échanger. (Dans une application réelle, il pourrait également contenir des informations de sécurité, les noms de l'acheteur et du courtier et bien d'autres choses. On pourrait également représenter ces données annexes dans d'autres objets.) Voici à quoi ressemblerait un fragment de schéma XML pour un ordre en bourse :

```
<element name="OrdreBoursier" type="OrdreBoursier"/>
<complexType name="OrdreBoursier">
    <element name="symbole" type="xsd:string"/>
    <element name="achat" type="xsd:boolean"/>
    <element name="nbparts" type="xsd:int"/>
</complexType>
```

Créons une classe Java personnalisée représentant un ordre boursier, classe que nous appellerons `OrdreBoursier_CoteClient`. Je devrais normalement appeler cette classe `OrdreBoursier`, mais je veux clairement montrer que je l'utiliserai dans la partie cliente de l'exemple. Nous écrirons une classe similaire côté serveur qui possèdera un nom équivalent. Je fais cela pour bien vous montrer que vous n'êtes pas obligés d'utiliser la même classe au deux extrémités de la transaction. Il ne serait en fait pas très judicieux d'utiliser la même classe ni même, dans la plupart des cas, possible.

`OrdreBoursier_CoteClient` possède trois propriétés en lecture/écriture nommées `Symbole`, `Achat` et `NbParts`, qui représentent respectivement le symbole de l'action, l'indicateur d'achat/vente et le nombre de parts à échanger.

```
package javasoap.livre.ch5;
public class OrdreBoursier_CoteClient {
   String   _symbole;
   boolean  _achat;
   int      _nbParts;
   public OrdreBoursier_CoteClient() {
   }
   public OrdreBoursier_CoteClient(String symbole,
                    boolean achat, int nbParts) {
      _symbole = symbole;
      _achat = achat;
```

3. JavaBeans donne d'autres moyens pour cela mais ceux-ci vont au-delà du sujet de ce livre.

```
        _nbParts = nbParts;
    }
    public String getSymbole() {
        return _symbole;
    }
    public void setSymbole(String symbole) {
        _symbole = symbole;
    }
    public boolean isAchat() {
        return _achat;
    }
    public void setAchat(boolean achat) {
        _achat = achat;
    }
    public int getNbParts() {
        return _nbParts;
    }
    public void setNbParts(int nbParts) {
        _nbParts = nbParts;
    }
}
```

Créons maintenant la classe `OrdreBoursier_CoteServeur` qui représentera l'ordre boursier côté serveur. Pour nous assurer que cette classe est différente de sa contrepartie cliente, éliminons le constructeur recevant les paramètres. Et pour faire bonne mesure, changeons également les noms des variables de la classe et leur ordre d'apparition.

```
package javasoap.livre.ch5;
public class OrdreBoursier_CoteServeur {
    int      _parts;
    boolean  _ordreAchat;
    String   _action;
    public OrdreBoursier_CoteServeur() {
    }
    public String getSymbole() {
        return _action;
    }
    public void setSymbole(String action) {
        _action = action;
    }
    public boolean isAchat() {
        return _ordreAchat;
    }
    public void setAchat(boolean ordreAchat) {
        _ordreAchat = ordreAchat;
    }
    public int getNbParts() {
        return _parts;
    }
    public void setNbParts(int parts) {
        _parts = parts;
```

```
    }
}
```

Nous pouvons ajouter une nouvelle méthode au service `urn:ServiceBoursierDeBase` nommée `executeOrdreBoursier()`, recevant un ordre boursier en paramètre. La valeur de retour de cette méthode est une chaîne de caractère décrivant l'ordre. Voici la classe `ServiceBoursierDeBase` modifiée. Nous pouvons tirer parti de la méthode `executeOrdre()` déjà présente dans cette classe. Dans la nouvelle méthode `executeOrdreBoursier()`, nous construisons un tableau `Object` à partir des trois propriétés du paramètre ordre et nous passons ce tableau à la méthode `executeOrdre()`.

```
package javasoap.livre.ch5;
public class ServiceBoursierDeBase {

   public ServiceBoursierDeBase() {
   }
   public String executeOrdreBoursier(OrdreBoursier_CoteServeur ordre) {
      Object[] params = new Object[3];
      params[0] = ordre.getSymbole();
      params[1] = new Integer(ordre.getNbParts());
      params[2] = new Boolean(ordre.isAchat());
      return executeOrdre(params);
   }
   public String[] getLaPlusActive() {

      // renvoie les actions les plus échangées
      String[] actives = { "ABC", "DEF", "GHI", "JKL" };
      return actives;
   }
   public int getVolumeTotal(String[] actions) {

      // récupère le volume de chaque action
      // depuis une source de données et renvoie le total

      int total = 345000;
      return total;
   }
   public String executeOrdre(Object[] params) {
      String resultat;
      try {
         String action = (String)params[0];
         Integer nbParts = (Integer)params[1];
         Boolean achat = (Boolean)params[2];
         String typeOrdre = "Achat";
         if (false == achat.booleanValue()) {
            typeOrdre = "Vente";
         }
         resultat = (typeOrdre + " de " + nbParts + " " + action);
      }
      catch (ClassCastException e) {
         resultat = "Type de paramètre incorrect";
```

```
        }
        return resultat;
    }
}
```

Pour déployer le service, nous avons besoin de faire correspondre le type personnalisé avec la classe Java qui l'implémente. Nous devons donner un nom au type et le qualifier par un espace de noms approprié. Ceci n'est pas différent de la méthode que nous avons utilisé pour déclarer un service. Cette correspondance se trouve au niveau du descripteur de déploiement, dans la section `isd:mappings`. Voici le descripteur que nous allons utiliser pour déployer le service sous Apache SOAP :

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:ServiceBoursierDeBase">
  <isd:provider
     type="java"
     scope="Application"
     methods="getVolumeTotal getLaPlusActive executeOrdre executeOrdreBoursier">
    <isd:java
       class="javasoap.livre.ch5.ServiceBoursierDeBase"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:ServiceBoursierDeBase" qname="x:OrdreBoursier"
       javaType="javasoap.livre.ch5.OrdreBoursier_CoteServeur"
       java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
       xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

Cette correspondance est importante et mérite donc d'être résumée avant de poursuivre. La correspondance d'un type personnalisé vers une classe Java est associée à un style d'encodage ainsi qu'à un nom de type pleinement qualifié. Une classe Java implémentant le type personnalisé est spécifiée, ainsi que les classes utilitaires utilisées pour la sérialisation et la désérialisation. Examinons maintenant les détails de l'exemple.

L'essentiel du descripteur de déploiement doit vous sembler familier. Il est similaire à ceux vus au chapitre 4. La différence se trouve dans l'entrée située dans la section `isd:mappings`. L'entrée `isd:map` décrit la correspondance du type OrdreBoursier avec la classe `OrdreBoursier_CoteServeur`. L'élément `isd:map` ne contient pas de données. Toute l'information est fournie sous forme d'attributs. Le premier, `encodingStyle`, spécifie le style d'encodage associé à la sérialisation du type. Un identifiant d'espace de noms, `x` possédant la valeur `urn:ServiceBoursierDeBase` est ensuite indiqué. J'utilise le nom du service comme qualificateur d'espace de noms pour le type personnalisé. Vous n'êtes toutefois pas obligé d'en faire autant et en fait, ne le voudrez probablement pas dans la plupart des cas. Si par exemple, vos types personnalisés sont utilisés par plu-

sieurs services ou si vous utilisez des types personnalisés dont les définitions sont spécifiées par des tiers, vous voudrez certainement qualifier le type personnalisé en utilisant un espace de noms approprié. L'attribut suivant, qname, spécifie le nom pleinement qualifié du type mis en correspondance. La valeur assignée est `x:OrdreBoursier`, qui est le même espace de noms `OrdreBoursier` qualifié à l'aide du nom de service. L'attribut `javaType` spécifie la classe Java locale au serveur, utilisée pour implémenter le type. Côté serveur, nous utilisons `javasoap.livre.ch5.OrdreBoursier_CoteServeur`. Les deux derniers attributs, `java2XMLClassName` et `xml2JavaClassName`, indiquent à Apache SOAP quelles classes Java locales sont à utiliser pour effectuer respectivement la sérialisation et la désérialisation. Apache SOAP possède une classe de sérialisation/désérialisation personnalisée qui peut effectuer la conversion entre des types XML personnalisés et des classes Java se conformant au modèle JavaBeans d'accesseur à une propriété. Elle peut prendre en charge à la fois la sérialisation et la désérialisation, ce qui est la raison pour laquelle nous l'utilisons pour les deux attributs. Au chapitre suivant, nous verrons comment créer des sérialiseurs personnalisés.

Maintenant que nous sommes en possession d'un descripteur de déploiement, nous pouvons progresser et redéployer `urn:ServiceBoursierDeBase`. Ceci fait, notre service est prêt à recevoir les invocations de la méthode `executeOrdreBoursier`. Nous devons toutefois effectuer un peu de paramétrage également dans l'application cliente. Examinons-la :

```
package javasoap.livre.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
public class ClientOrdreBoursier
{
  public static void main(String[] args) throws Exception
  {
    URL url = new
       URL("http://localhost:8080/soap/servlet/rpcrouter");
    Call appel = new Call();
    SOAPMappingRegistry smr = new SOAPMappingRegistry();
    appel.setSOAPMappingRegistry(smr);
    appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    appel.setTargetObjectURI("urn:ServiceBoursierDeBase");
    appel.setMethodName("executeOrdreBoursier");
    BeanSerializer beanSer = new BeanSerializer();
    // Effectue la correspondance du type ordre boursier
    smr.mapTypes(Constants.NS_URI_SOAP_ENC,
       new QName("urn:ServiceBoursierDeBase", "OrdreBoursier"),
       OrdreBoursier_CoteClient.class, beanSer, beanSer);
    // Crée une instance de l'ordre boursier
    OrdreBoursier_CoteClient ordre =
          new OrdreBoursier_CoteClient("XYZ", false, 350);
```

```
        Vector params = new Vector();
        params.addElement(new Parameter("ordre",
                          OrdreBoursier_CoteClient.class, ordre, null));
        appel.setParams(params);
        Response rep;
        try {
          rep = appel.invoke(url, "");
          Parameter ret = rep.getReturnValue();
          Object desc = ret.getValue();
          System.out.println("Description de l'ordre : " + desc);
        }
        catch (SOAPException e) {
          System.err.println("SOAPException (" +
                             e.getFaultCode() + "): " +
                             e.getMessage());
        }
      }
    }
```

Après la création de l'objet `Call`, nous instancions `org.apache.soap.encoding.SOAPMappingRegistry` sous le nom `smr`. Cette classe contient les correspondances des types personnalisés vers les classes Java et est passée à l'objet `Call` à l'aide de la méthode `setSOAPMappingRegistry()` dudit objet `Call`. Nous n'avions pas eu à faire cela dans nos exemples précédents car l'objet `Call` crée un registre de correspondance pour lui-même si aucun ne lui est envoyé. Le `SOAPMappingRegistry()` contient toutes les correspondances prédéfinies, comme par exemple celles dont nous voulons tirer parti pour les tableaux. Nous y ajouterons notre correspondance sous peu.

Nous appelons ensuite la méthode `call.setEncodingStyleURI()`. Cette méthode spécifie le style d'encodage à utiliser pour les paramètres de type personnalisé. La constante `NS_URI_SOAP_ENC` de la classe `org.apache.soap.Constants`, représente l'espace de noms `http://schemas.xmlsoap.org/soap/encoding/` que nous avons utilisé jusqu'à présent. Il doit s'agir du même espace de noms de style d'encodage que celui que nous avons spécifié dans le descripteur de déploiement pour le type `OrdreBoursier`. Les méthodes `setTargetObjectURI()` et `setMethodName()` sont utilisées de la même façon que dans les précédents exemples. L'étape suivante consiste à instancier `org.apache.soap.encoding.soapenc.BeanSerializer`. Nous sommes en mesure d'utiliser ce sérialiseur standard car notre type personnalisé se conforme au modèle JavaBeans d'accesseur à une propriété. Nous pouvons maintenant établir la correspondance en appelant `smr.mapTypes()`. Le premier paramètre est `Constants.NS_URI_SOAP_ENC`; il spécifie que le style d'encodage utilisé pour cette correspondance est l'encodage SOAP standard. Vous vous demandez peut-être pourquoi nous avons besoin de cela puisque nous avons déjà spécifié le style d'encodage pour l'appel entier un peu plus tôt. La raison est simple : ce paramètre vous donne l'opportunité de supplanter le style d'encodage utilisé pour cette correspondance particulière. Si toutefois vous utilisez `null` comme valeur du paramètre, vous obtiendrez une correspondance qui tentera d'utiliser l'espace de noms `null` pour le style d'encodage, ce qui est incorrect. En examinant l'enveloppe SOAP de ce message, vous verrez que dans la mesure où le style d'encodage est le même que celui de l'ensemble de l'appel, l'attribut `encodingStyle` n'a pas besoin d'être répété pour ce

paramètre sérialisé. Si le style d'encodage n'avait pas été le même que celui de `Call`, il serait apparu comme un attribut du paramètre.

Le paramètre suivant de `smr.mapTypes()` est une instance de `org.apache.soap.util.xml.Qname` qui représente un nom pleinement qualifié (c'est-à-dire incluant un qualificateur d'espace de noms suivi par un nom). Nous utilisons `urn:ServiceBoursierDeBase` comme espace de noms et `OrdreBoursier` comme nom. Le troisième paramètre est `OrdreBoursier_CoteClient.class`, la classe Java qui implémente le type personnalisé. Les deux paramètres suivants sont les instances du sérialiseur et du désérialiseur qui seront utilisés pour ce type. Nous utilisons l'objet `beanSer` que nous avons créé plus tôt pour les deux, la classe `org.apache.soap.encoding.soapenc.BeanSerializer` implémentant à la fois la sérialisation et la désérialisation.

Le reste est assez simple. Nous créons une instance d'`OrdreBoursier_CoteClient` en tirant parti du constructeur paramétré pour positionner les valeurs de ses propriétés. Nous initialisons ensuite un `Vector` de paramètres, comme nous l'avons fait dans les précédents exemples. En exécutant cet exemple, vous devriez obtenir le résultat suivant:

```
Description de l'ordre : Vente de 350 XYZ
```

Examinons l'enveloppe SOAP transmise. La partie significative commence par l'élément `ordre`, qui représente le paramètre de type personnalisé passé à la méthode de service `executeOrdreBoursier`. La valeur assignée à `xsi:type` est `ns1:OrdreBoursier`. L'identifiant d'espace de nom `ns1` est déclaré comme étant `urn:ServiceBoursierDeBase` dans l'élément `executeOrdreBoursier` parent. Et `OrdreBoursier` est le nom spécifié pour notre type personnalisé. On trouve trois éléments enfants de l'élément `ordre`, chacun d'eux correspondant aux propriétés du type personnalisé `OrdreBoursier`. Le nom de chaque élément correspond exactement au nom de la propriété. Ceci est crucial, Apache SOAP utilisant la réflexion Java pour retrouver l'ensemble des méthodes de la classe Java associée. De ce fait, les noms dans l'enveloppe doivent exactement correspondre à ceux utilisés par la classe. Chacun de ces éléments propriété est explicitement typé et ces types doivent également coïncider avec les types des propriétés correspondantes.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:executeOrdreBoursier xmlns:ns1="urn:ServiceBoursierDeBase"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <ordre xsi:type="ns1:OrdreBoursier">
         <nbParts xsi:type="xsd:int">350</nbParts>
         <achat xsi:type="xsd:boolean">false</achat>
         <symbole xsi:type="xsd:string">XYZ</symbole>
      </ordre>
     </ns1:executeOrdreBoursier>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Voyons maintenant comment les types personnalisés sont pris en charge sous GLUE et utilisons pour cela la classe `ServiceBoursierDeBase` sans modification. Nous avons be-

soin d'ajouter la méthode executeOrdreBoursier() à l'interface IServiceBoursierDeBase. Nous pouvons déployer le service comme nous l'avons fait auparavant. Voici la nouvelle version de IServiceBoursierDeBase:

```
package javasoap.livre.ch5;
public interface IServiceBoursierDeBase {
  int getVolumeTotal(String[] symboles);
  String executeOrdre(Object[] params);
  String[] getLaPlusActive();
  String executeOrdreBoursier(OrdreBoursier_CoteClient ordre);
}
```

L'écriture d'applications accédant à des services par l'API GLUE a été jusque-là aisée en raison du fait que nous n'avons pas eu à nous occuper directement des constructions SOAP. Nous devrions pouvoir continuer ainsi, mais nous allons cependant rencontrer un problème. La méthode executeOrdreBoursier() de l'interface IServiceBoursierDeBaseInterface considère que le paramètre est une instance de OrdreBoursier_CoteClient. Mais la méthode executeOrdreBoursier de la classe ServiceBoursierDeBase qui implémente le service utilise un paramètre de OrdreBoursier_CoteServeur. Il y a donc une discordance due à la conception même. Par défaut, GLUE recherche la même classe côté client et serveur. Si nous avions utilisé la classe OrdreBoursier_CoteServeur dans notre application cliente au lieu de la classe OrdreBoursier_CoteClient, tout aurait fonctionné parfaitement. Nous n'allons pas le faire et croyez-moi simplement si je vous dis que cela fonctionne (ou mieux encore, essayez par vous-même). Nous allons devoir adopter une autre approche.

Lorsque je développe des systèmes distribués, j'apprécie de partager les interfaces Java entre le code client et le code serveur. Je préfère toutefois ne pas partager les classes Java, en particulier si ces classes contiennent du code spécifique au client ou au serveur. Dans cet exemple, nous pourrions revoir la conception de nos classes d'ordres boursiers pour qu'elles contiennent les données significatives sans aucune des fonctionnalités du client ou du serveur. Nous voulons que le client comme le serveur utilisent la classe d'un même package et donc, en tant qu'implémenteur du service d'ordres boursiers, nous devrions distribuer le package contenant cette classe aux applications clientes des développeurs. Cela devrait fonctionner et n'est pas une pratique exceptionnelle. En revenant à cette notion de partage d'interfaces Java plutôt que de classes, nous pourrions créer une interface pour les données de l'ordre et faire implémenter cette interface à la fois par les classes du client et du serveur. C'est une façon élégante et propre d'effectuer un partage entre les codes serveur et client sans toutefois partager de code exécutable. Encore une fois, cette interface devra se trouver dans un package accessible à la fois aux systèmes serveur et client. Ce mécanisme devrait être disponible dans une prochaine version de GLUE (qui est peut-être déjà disponible au moment où vous lisez ces lignes).

GLUE possède un autre mécanisme permettant de faire cela. Ce mécanisme ne vous oblige pas à modifier la partie serveur et le travail nécessaire côté client est trivial. Nous allons créer un nouveau package pour la partie client de cet exemple, car nos travaux nous amèneraient à créer des fichiers ayant les mêmes noms que ceux que nous avons créés plus tôt. Ce nouveau package nous évitera d'écraser les fichiers si nous exécutons le client et le serveur sur la même machine. Souvenez-vous donc que cet exemple fait partie d'un nouveau package appelé javasoap.livre.ch5.client et que les fichiers que

nous allons développer doivent se trouver dans le répertoire correspondant à ce package.

J'ai intentionnellement évité jusqu'à présent la discussion sur WSDL, bien que GLUE soit entièrement basé dessus. Cependant, la première étape de ce processus fait usage de l'utilitaire GLUE `wsdl2java`, qui génère du code Java basé sur un fichier WSDL. D'où vient WSDL alors ? Le serveur GLUE le génère automatiquement. Les applications côté client basées sur l'API GLUE en ont constamment tiré profit. `wsdl2java` génère l'interface Java pour la liaison au service, une classe technique, une classe Java contenant une structure de données représentant les champs du type personnalisé que nous utilisons et un fichier de correspondance. Celui-ci est essentiellement une définition de schéma pour le type personnalisé. Il indique à GLUE comment effectuer la correspondance entre la classe Java de structure de données et les champs du type personnalisé. GLUE utilise plusieurs mécanismes pour prendre en charge cette correspondance. Dans notre cas, le fichier définit explicitement une correspondance plutôt que de la baser sur la réflexion Java.

Continuons et générons les fichiers de l'application cliente. Entrez la commande suivante après vous être assuré que vous vous trouvez dans le répertoire du package `javasoap.livre.ch5.client` :

```
wsdl2java http://localhost:8004/glue/urn:ServiceBoursierDeBase.wsdl
   -p javasoap.livre.ch5.client
```

Le paramètre à passer à l'utilitaire `wsdl2java` est l'URL complète du service WSDL, comme lorsque nous avons utilisé les méthodes `bind()` dans les exemples précédents. L'option `-p` indique à l'utilitaire le nom du package local pour lequel le code doit être généré : `javasoap.livre.ch5.client`. Le résultat fourni par `wsdl2java` consiste en quatre fichiers. Le premier est *IServiceBoursierDeBase.java*, qui contient l'interface Java nécessaire à la liaison au service. Nous l'avions écrite à la main jusqu'à présent. Examinons celle générée par `wsdl2java` :

```
// généré par GLUE
package javasoap.livre.ch5.client;
public interface IServiceBoursierDeBase
  {
  String executeOrdreBoursier( OrdreBoursier_CoteServeur arg0 );
  String[] getLaPlusActive();
  int getVolumeTotal( String[] arg0 );
  String executeOrdre( Object[] arg0 );
  }
```

Les seules différences entre cette définition d'interface et celle que nous avions écrit nous-mêmes sont une déclaration de package différente, le nom du paramètre de méthode et le fait qu'`executeOrdreBoursier` reçoit un paramètre d'`OrdreBoursier_CoteServeur` au lieu d'`OrdreBoursier_CoteClient`. Souvenez-vous que la précédente version de `IServiceBoursierDeBase` ne fonctionnerait pas car GLUE utilise par défaut le même nom de classe côté client et serveur. À première vue, l'utilisation de la classe `OrdreBoursier_CoteServeur` semble contradictoire avec notre souhait de découpler le code du serveur du code du client. Mais souvenez-vous que la déclaration de package est `javasoap.livre.ch5.client` et que cette interface Java ne référence donc pas la même classe `OrdreBoursier_CoteServeur` que celle utilisée par no-

tre serveur et qui appartient à un package différent. Voyons à quoi ressemble la classe `OrdreBoursier_CoteServeur` générée par `wsdl2java` et située dans le fichier *OrdreBoursier_CoteServeur.java* :

```
// généré par GLUE
package javasoap.livre.ch5.client;
public class OrdreBoursier_CoteServeur
  {
  public int _parts;
  public boolean _ordreAchat;
  public String _action;
  }
```

Ce n'est qu'une image de la classe du package `javasoap.livre.ch5` correspondante, utilisée côté serveur. Elle reflète cependant correctement les valeurs des données. Nous allons créer dans notre application cliente, une instance de cette classe à passer au service, puisqu'il s'agit du type attendu par la méthode `executeOrdreBoursier()` de l'interface `IServiceBoursierDeBase` qui a été générée.

Le troisième fichier généré par `wsdl2java` est *ServiceBoursierDeBase.map*, qui contient le schéma de correspondance mentionné ci-dessus. Vous pourrez étudier le contenu du fichier de correspondance par vous-même. Contentons-nous de dire qu'il contient des entrées XML qui définissent la correspondance des champs entre Java côté client et Java côté serveur. Le dernier fichier généré est *ServiceBoursierDeBaseHelper.java*, qui contient une classe technique permettant d'effectuer l'opération `bind()`. Je ne l'utilise pas et nous l'ignorerons donc.

Comme il serait bien plus long de décrire le processus que de l'effectuer, lançons-nous dans l'écriture de l'application cliente :

```
package javasoap.livre.ch5.client;
import electric.registry.RegistryException;
import electric.registry.Registry;
import electric.xml.io.Mappings;
public class ClientOrdreBoursier2 {
   public static void main(String[] args) throws Exception
   {
      try {
        Mappings.readMappings("ServiceBoursierDeBase.map");
        IServiceBoursierDeBase srv =
          (IServiceBoursierDeBase)Registry.bind(
          "http://localhost:8004/glue/urn:ServiceBoursierDeBase.wsdl",
          IServiceBoursierDeBase.class);
        OrdreBoursier_CoteServeur ordre =
                      new OrdreBoursier_CoteServeur();
        ordre._action = "MINDSTRM";
        ordre._ordreAchat = true;
        ordre._parts = 500;
        String desc = srv.executeOrdreBoursier(ordre);
        System.out.println("Description de l'ordre : " + desc);
     }
     catch (RegistryException e)
```

```
      {
         System.out.println(e);
      }
    }
  }
```

L'étape critique consiste à lire les correspondances avant d'effectuer l'opération de liaison. Ceci se fait en appelant la méthode `readMappings()` de la classe `electric.xml.io.Mappings` fournie avec GLUE. Après l'appel `bind()`, nous créons tout simplement une instance d'`OrdreBoursier_CoteServeur`, renseignons ses champs et appelons la méthode `executeOrdreBoursier`. Le résultat de l'exécution de cette application doit être :

```
Description de l'ordre : Achat de 500 MINDSTRM
```

Examinons maintenant l'enveloppe SOAP générée par cette application. Comme nous pouvions nous y attendre, l'élément `executeOrdreBoursier` est qualifié par un espace de noms utilisant le nom du service. L'élément enfant `arg0` possède un élément sérialisé séparément et nommé `id0`, comme celui généré par GLUE pour un paramètre tableau. Cet exemple illustre pourquoi GLUE à généré l'identifiant d'espace de noms `ns2` qui semble référencer le package dans lequel se trouve la classe côté serveur. Cet espace de noms est utilisé ici pour qualifier la valeur de l'attribut `xsi:type` à l'aide d'une valeur correspondant au nom de la classe implémentant `OrdreBoursier_CoteServeur`. Les éléments enfants de l'élément `id0` contiennent les champs destinés au type personnalisé.

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:executeOrdreBoursier xmlns:n='urn:ServiceBoursierDeBase'>
        <arg0 href='#id0'/>
      </n:executeOrdreBoursier>
      <id0 id='id0' soapenc:root='0'
       xmlns:ns2='http://www.themindelectric.com/package/javasoap.livre.ch5/'
       xsi:type='ns2:OrdreBoursier_CoteServeur'>
        <_parts xsi:type='xsd:int'>500</_parts>
        <_ordreAchat xsi:type='xsd:boolean'>true</_ordreAchat>
        <_action xsi:type='xsd:string'>MINDSTRM</_action>
      </id0>
    </soap:Body>
</soap:Envelope>
```

Renvoyer des types personnalisés

Il est également utile (et courant) de renvoyer des types personnalisés à partir d'appels à des méthodes de service. Nous pouvons améliorer notre service boursier en offrant une méthode qui recevra un symbole d'action en paramètre et renverra son cours le plus haut et le plus bas de la journée. Les classes `MinMax_CoteServeur` et

MinMax_CoteClient représentent respectivement les cours maximum et minimum sur le serveur et le client.

```
package javasoap.livre.ch5;
public class MinMax_CoteServeur {
   public float _max;
   public float _min;
   public MinMax_CoteServeur() {
   }
   public MinMax_CoteServeur (float max, float min) {
      setMax(max);
      setMin(min);
   }
   public float getMax() {
      return _max;
   }
   public void setMax(float max) {
      _max = max;
   }
   public float getMin() {
      return _min;
   }
   public void setMin(float min) {
      _min = min;
   }
}
```

```
package javasoap.livre.ch5;
public class MinMax_CoteClient {
   public float _max;
   public float _min;
   public String toString() {
      return "Min : " + _min +
        " Max : " + _max;
   }
   public MinMax_CoteClient() {
   }
   public float getMax() {
      return _max;
   }
   public void setMax(float max) {
      _max = max;
   }
   public float getMin() {
      return _min;
   }
   public void setMin(float min) {
      _min = min;
   }
}
```

La classe serveur inclut un constructeur paramétré pour faciliter la création de la valeur de retour. La classe client comprend une méthode `toString()` pour permettre à notre application d'afficher facilement le contenu de l'objet renvoyé du serveur. Ajoutons une nouvelle méthode à la classe `ServiceBoursierDeBase`, appelée `getMinMax()`. Cette méthode reçoit un unique paramètre chaîne représentant le symbole de l'action et renvoie une instance de `MinMax_CoteServeur`. Voici la classe avec cette nouvelle méthode, le code inchangé étant omis :

```
package javasoap.livre.ch5;
public class ServiceBoursierDeBase {
   public ServiceBoursierDeBase() {
   }
   . . .
   . . .
   public MinMax_CoteServeur getMinMax(String action) {

      // renvoie les valeurs min et max de l'action spécifiée
      return new MinMax_CoteServeur((float)109.5,
                     (float)110.375);
   }
}
```

Pour rendre cette nouvelle méthode disponible sous Apache SOAP, nous allons devoir redéployer le service en utilisant un descripteur de déploiement modifié. Nous devons ajouter `getMinMax` à la liste des méthodes et ajouter une entrée dans la section des correspondances pour l'objet min/max. La seconde entrée de la correspondance définit le type personnalisé `MinMax`, qualifié par un espace de noms à l'aide du nom de service `urn:ServiceBoursierDeBase`. Voici le descripteur de déploiement modifié :

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:ServiceBoursierDeBase">
  <isd:provider
     type="java"
     scope="Application"
     methods="getVolumeTotal getLaPlusActive executeOrdre executeOrdreBoursier
              getMinMax">
    <isd:java
       class="javasoap.livre.ch5.ServiceBoursierDeBase"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:ServiceBoursierDeBase" qname="x:OrdreBoursier"
       javaType="javasoap.livre.ch5.OrdreBoursier_CoteServeur"
       java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
       xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
```

```
    <isd:map
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:x="urn:ServiceBoursierDeBase" qname="x:MinMax"
      javaType="javasoap.livre.ch5.MinMax_CoteServeur"
      java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
      xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

Nous pouvons maintenant créer une application cliente qui invoque la méthode de service getMinMax et reçoit un objet min/max en retour. Pour le client Apache SOAP, nous utiliserons la classe MinMax_CoteClient pour représenter la valeur de retour. Voici la nouvelle application :

```
package javasoap.livre.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
public class MinMaxClient
{
  public static void main(String[] args) throws Exception
  {
    URL url = new
      URL("http://localhost:8080/soap/servlet/rpcrouter");
    Call appel = new Call();
    SOAPMappingRegistry smr = new SOAPMappingRegistry();
    appel.setTargetObjectURI("urn:ServiceBoursierDeBase");
    appel.setMethodName("getMinMax");
    appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    appel.setSOAPMappingRegistry(smr);
    BeanSerializer beanSer = new BeanSerializer();
    // Effectue la correspondance du type Min/Max
    smr.mapTypes(Constants.NS_URI_SOAP_ENC,
            new QName("urn:ServiceBoursierDeBase", "MinMax"),
            MinMax_CoteClient.class, beanSer, beanSer);
    String action = "XYZ";
    Vector params = new Vector();
    params.addElement(new Parameter("action",
                          String.class, action, null));
    appel.setParams(params);
    Response rep;
    try {
      rep = appel.invoke(url, "");
      Parameter ret = rep.getReturnValue();
      MinMax_CoteClient minmax =
            (MinMax_CoteClient)ret.getValue();
      System.out.println(minmax);
```

```
      }
      catch (SOAPException e) {
        System.err.println("SOAPException (" +
                           e.getFaultCode() + "): " +
                           e.getMessage());
      }
    }
  }
```

`smr.MapTypes()` effectue la correspondance du type personnalisé `MinMax` vers la classe Java `MinMax_CoteClient`. Comme auparavant, nous pouvons utiliser le `BeanSerializer` d'Apache pour faire la conversion entre XML et Java dans la mesure où notre classe est conforme au modèle JavaBeans d'accesseur à une propriété. Nous positionnons un paramètre `String` nommé `action` à passer à la méthode `getMinMax` (bien que nous ne l'utilisions pas en réalité dans le code serveur). Une fois la méthode invoquée, nous transtypons la valeur de retour de `rep.getReturnValue()` en une instance de `MinMax_CoteClient`. Nous passons alors la variable paramètre de retour, `ret`, à la méthode `System.out.println()` pour affichage. C'est tout ce qu'il nous faut car nous avons implémenté la méthode `toString()` dans notre classe `MinMax_CoteClient`. En exécutant cet exemple, vous obtiendrez le résultat suivant :

```
Min : 109.5 Max : 110.375
```

Voici l'enveloppe SOAP renvoyée à l'invocation de cette méthode. L'élément `return` est de type `MinMax`, qualifié par l'espace de noms `urn:ServiceBoursierDeBase`. Les propriétés, qui constituent les éléments enfants de l'élément `return`, sont typées en `float` et apparaissent avec leurs valeurs correspondantes.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <SOAP-ENV:Body>
    <ns1:getMinMaxReponse
      xmlns:ns1="urn:ServiceBoursierDeBase"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="ns1:MinMax">
         <min xsi:type="xsd:float">109.5</min>
         <max xsi:type="xsd:float">110.375</max>
      </return>
   </ns1:getMinMaxReponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Pour renvoyer un `MinMax` avec GLUE, nous allons suivre le même chemin que pour passer des types personnalisés. Une fois encore nous allons ranger notre travail côté client dans un autre package, `javasoap.livre.ch5.client`. Redémarrez simplement l'application `AppBoursiereDeBase` pour que le service soit déployé. Exécutez maintenant l'utilitaire `wsdl2java` depuis le répertoire correspondant au package `javasoap.livre.ch5.client` :

```
wsdl2java http://localhost:8004/glue/urn:ServiceBoursierDeBase.wsdl
    -p javasoap.livre.ch5.client
```

Le code suivant constitue la nouvelle interface IServiceBoursierDeBase générée par wsdl2java. La méthode getMinMax() renvoie une instance de MinMax_CoteServeur. Souvenez-vous que cette classe est la nouvelle générée par GLUE comme composant du package javasoap.livre.ch5.client et non celle utilisée par notre classe serveur AppBoursiereDeBase.

```
// généré par GLUE
package javasoap.livre.ch5.client;
public interface IServiceBoursierDeBase
  {
  MinMax_CoteServeur getMinMax( String arg0 );
  String executeOrdreBoursier( OrdreBoursier_CoteServeur arg0 );
  String[] getLaPlusActive();
  int getVolumeTotal( String[] arg0 );
  String executeOrdre( Object[] arg0 );
  }
```

La classe suivante, MinMax_CoteServeur, est simplement une structure de données Java reflétant les champs qui seront mis en correspondance avec le type personnalisé MinMax. J'ai ajouté la méthode toString() à la main pour rendre l'affichage des résultats plus simple.

```
package javasoap.livre.ch5.client;
public class MinMax_CoteServeur {
  public float _max;
  public float _min;
  public String toString() {
     return "Min : " + _min +
        " Max : " + _max;
  }
}
```

Créons maintenant l'application cliente sous GLUE qui invoquera la méthode de service getMinMax et affichera le contenu de la valeur de retour obtenue. Il n'y a vraiment pas grand-chose à faire. Nous avons simplement à lire le fichier de correspondance, à effectuer la liaison puis à appeler la méthode getMinMax par l'interface liée. L'instance de javasoap.livre.ch5.client.MinMax_CoteServeur résultante est passée à System.out.println() pour l'affichage. Tout comme dans l'exemple Apache SOAP, la méthode toString() de la classe prend en charge la création de la chaîne à afficher.

```
package javasoap.livre.ch5.client;
import electric.registry.RegistryException;
import electric.registry.Registry;
import electric.xml.io.Mappings;
public class MinMaxClient2 {
   public static void main(String[] args) throws Exception
   {
      try {
        Mappings.readMappings("ServiceBoursierDeBase.map");
        IServiceBoursierDeBase srv = (IServiceBoursierDeBase)Registry.bind(
          "http://localhost:8004/glue/urn:ServiceBoursierDeBase.wsdl",
          IServiceBoursierDeBase.class);
```

```
            MinMax_CoteServeur minmax = srv.getMinMax("TOUT");
            System.out.println(minmax);
        }
        catch (RegistryException e)
        {
           System.out.println(e);
        }
     }
  }
```

Voici l'enveloppe SOAP renvoyée par le serveur. Vous devriez maintenant être capable de bien comprendre son contenu. GLUE utilise une référence à une instance du type personnalisé sérialisée séparément, tout comme il le fait lorsqu'il renvoie un tableau.

```
<soap:Envelope
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
 soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:getMinMaxResponse xmlns:n='urn:ServiceBoursierDeBase'>
        <Result href='#id0'/>
      </n:getMinMaxResponse>
      <id0 id='id0' soapenc:root='0'
 xmlns:ns2='http://www.themindelectric.com/package/javasoap.livre.ch5/'
        xsi:type='ns2:MinMax_CoteServeur'>
        <_min xsi:type='xsd:float'>109.5</_min>
        <_max xsi:type='xsd:float'>110.375</_max>
      </id0>
    </soap:Body>
</soap:Envelope>
```

Dans ce chapitre, nous avons traité en profondeur de l'utilisation des tableaux et des types de données personnalisés et de la façon dont ces structures sont supportées par Apache SOAP et GLUE. Il existe d'autres types utiles supportés par ces implémentations SOAP ou d'autres. Ces types peuvent inclure les `Vector` Java et les classes `Hashtable`, les collections Java ainsi que plusieurs autres classes Java communément utilisées.

Nous avons vu qu'Apache SOAP et GLUE adoptaient une approche différente des types complexes et que tous deux les supportaient efficacement. Il arrivera toutefois que vous ayez besoin d'utiliser un type personnalisé qui ne peut ou ne devrait pas être sérialisé de la façon fournie par votre implémentation SOAP. Cette situation peut survenir tout simplement du fait que certains types de données particuliers, comme les tableaux multidimensionnels ou épars, ne sont pas supportés ou pour toute autre raison liée à votre application. Nous aborderons ce problème au chapitre suivant.

6

Sérialisation personnalisée

Vous pourriez considérer que vos services web et vos applications clientes fonctionnent parfaitement en utilisant uniquement des types de données supportés par votre implémentation SOAP. Après tout, les technologies que nous abordons fournissent une grande marge de flexibilité grâce au support des tableaux et des types personnalisés. Toutefois à un moment ou à un autre, vous aurez peut-être besoin de sérialiser quelque chose qui ne pourra l'être à l'aide des techniques vues jusqu'à présent. Il se peut que vous vouliez sérialiser les données d'une classe Java d'une façon qui soit plus cohérente avec vos applications existantes, même si leur encodage n'est pas standardisé. Vous pourriez aussi vouloir ajouter le support de constructions décrites dans les spécifications SOAP mais non supportées dans les implémentations disponibles.

L'implémentation de SOAP que vous utilisez peut vous obliger à écrire des sérialiseurs personnalisés pour certains types d'objets. Apache fournit par exemple un sérialiseur Bean. Il pourrait fournir d'autres sérialiseurs adaptés à des cas particuliers et n'était pas obligé de fournir celui-ci. Les conditions dans lesquelles vous aurez à fabriquer un sérialiseur personnalisé sont donc dépendantes de l'implémentation.

Il est également important de considérer l'utilité des données à sérialiser. La plupart des règles Java concernant l'implémentation de l'interface `Serializable` seront applicables ici. Ainsi, que l'on soit sous SOAP ou non, sérialiser un flux d'entrées/sorties n'est probablement pas quelque chose de très judicieux. Cela reste vrai pour toute sortes de systèmes distribués : vous devez comprendre le sens de vos données avant de les envoyer à l'aveuglette.

Nous traiterons quelques exemples de sérialisation personnalisée dans ce chapitre et explorerons les techniques et API disponibles à la fois sous Apache SOAP et GLUE. Contrairement aux chapitres précédents, nous ne présenterons pas les exemples sous chacune des deux technologies. Vous devez désormais être suffisamment familier d'Apache SOAP et de GLUE pour avoir une idée sur la façon de les utiliser.

Encodage de types personnalisés

Démarrons par un type de données personnalisé utilisant une sérialisation non standard. Nous allons développer cet exemple sous Apache SOAP. Imaginons que nous dé-

veloppons un service web agissant comme une passerelle vers une source de données du marché boursier. Nous voudrions envoyer au service nos données, préalablement sérialisées, dans la forme utilisée par la source. Eh oui, encore un exemple pas très évident ! Mais d'ici à ce que nous le terminions, je parie que vous aurez pensé à plusieurs utilisations de cette technique. Le service est conçu pour permettre aux applications clientes d'envoyer au service fournissant les données, des formats agréés, sans avoir à modifier le service web lui-même. Le format des données sérialisées n'est ainsi pas connu du service web. Il n'est compris que par l'application cliente et la source de données. La *figure 6-1* montre à quoi ressemblerait une telle architecture. Les applications clientes utilisent le même service et invoquent les mêmes méthodes. Elles peuvent toutefois utiliser ou non le même format de sérialisation pour les données envoyées à la source de données.

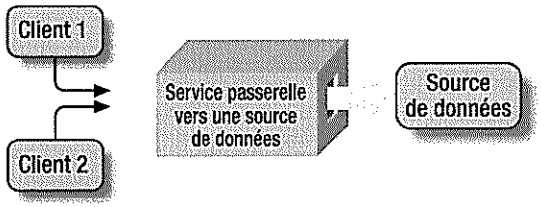


Figure 6-1. Service passerelle

Considérons que le format de données par défaut de la source est un format linéaire autodescriptif. Ce format est une chaîne de caractères de longueur arbitraire contenant un nombre variable de champs nommés ainsi que leurs valeurs chaînes. Le format peut accepter de 1 à 9 champs. Le premier caractère du message contient le nombre de champs et peut être n'importe quel caractère ASCII entre 1 et 9. Chaque champ est constitué du nom du champ suivi de deux points puis de la valeur du champ également suivie de deux points. Voici un exemple de message contenant deux champs nommés SYMBOLE et PRIX ainsi que leurs valeurs correspondantes ZZ et 110.5 :

```
2SYMBOLE:ZZ:PRIX:110.5:
```

Ce sera le format que nous utiliserons pour sérialiser les messages. Créons tout d'abord un nouveau service nommé `urn:ServiceSourceDeDonnees` contenant une méthode nommée `envoieMessage`. Cette méthode reçoit et renvoie un message en paramètre. Avant de définir la classe Java du service, il nous faut donc définir la classe Java du message. La classe `MessageSourceDeDonnees` représente le message à utiliser avec ce service. Elle contient une liste de paires noms/valeurs permettant de conserver la trace des noms de champs et de leurs valeurs.

```
package javasoap.livre.ch6;
import java.util.*;
public class MessageSourceDeDonnees {
   Hashtable _champs = new Hashtable();
   public MessageSourceDeDonnees() {
   }
   public void ajouteChamp(String nom, String valeur) {
      _champs.put(nom, valeur);
```

```
    }
    public Enumeration getNomsChamps() {
       return _champs.keys();
    }
    public String getValeurChamp(String nom) {
       return (String)_champs.get(nom);
    }
    public String toString() {
       String cnt = String.valueOf(_champs.size());
       StringBuffer msg = new StringBuffer();
       msg.append(cnt);
       msg.append(":");
       for (Enumeration e =_champs.keys(); e.hasMoreElements();) {
          String nom = (String)e.nextElement();
          String valeur = (String)_champs.get(nom);
          String section = nom + ":" + valeur + ":";
          msg.append(section);
       }
       return msg.toString();
    }
    public void analyseMessageFormate(String msg) {
       _champs.clear();
       StringTokenizer st = new StringTokenizer(msg, ":");
       String token = st.nextToken();
       int cnt = Integer.parseInt(token);
       for (int i = 0; i < cnt; i++) {
          String nom = st.nextToken();
          String valeur = st.nextToken();
          ajouteChamp(nom, valeur);
       }
    }
```

Les champs sont stockés dans la variable `_champs`, qui est une instance de `java.util.Hashtable`. La méthode `ajouteChamp()` permet d'ajouter un champ en fournissant en paramètres son nom et sa valeur qui sont ajoutés à la table de hachage en utilisant le nom comme clé. Les méthodes `getNomsChamps()` et `getValeurChamp()` donnent accès aux noms et aux valeurs des éléments de la table de hachage, sans que ne soit visible le fait que la classe utilise en interne `java.util.Hashtable`. Les deux dernières méthodes sont les plus intéressantes. La méthode `toString()` renvoie une chaîne de caractères contenant tous les champs ainsi que leurs valeurs dans le format de message que nous avons défini plus haut. La méthode `analyseMessageFormate()` reçoit en paramètre une chaîne au format défini et insère son contenu dans la table de hachage. Ces deux méthodes nous seront utiles lorsque nous voudrons sérialiser et désérialiser un type de message personnalisé.

Nous pouvons maintenant créer la classe Java pour ce nouveau service. Nous l'appelerons `ServiceSourceDeDonnees` et nous y incluerons une méthode nommée `envoieMessage()` recevant une instance de `MessageSourceDeDonnees` en paramètre. Cette méthode renverra également une instance de `MessageSourceDeDonnees`. Voici la classe `ServiceSourceDeDonnees` :

```
package javasoap.livre.ch6;
public class ServiceSourceDeDonnees {
  public ServiceSourceDeDonnees() {
  }
  public MessageSourceDeDonnees envoieMessage(MessageSourceDeDonnees msg) {
     // nous pourrions retirer le message mis en forme ici
     // en appelant msg.toString();
     String action = msg.getValeurChamp("SYMBOL");
     String requete = msg.getValeurChamp("DEMANDE");
    MessageSourceDeDonnees res = new MessageSourceDeDonnees();
    res.ajouteChamp("SYMBOLE", action);
    if (requete.equals("PRIX"))
         res.ajouteChamp("PRIX", "123.5");
      return res;
    }
 }
```

Nous ne faisons dans cet exemple que prendre le symbole de l'action dans le message source et l'utiliser comme valeur de retour, à laquelle on aura préalablement ajouté le champ prix comme si le champ nommé DEMANDE possédait la valeur PRIX. Ainsi la méthode `envoieMessage()` utilise certaines des données du paramètre d'entrée pour créer une valeur de retour du type `MessageSourceDeDonnees`.

Cette classe n'obéissant pas aux conventions de nommage JavaBeans, nous ne pouvons utiliser le `BeanSerializer` fourni par Apache SOAP pour sérialiser et désérialiser les instances du message source que nous avons créé. Nous allons donc créer notre propre sérialiseur. Une classe Java personnalisée destinée à sérialiser des objets doit implémenter l'interface `org.apache.soap.util.xml.Serializer`.

Pour implémenter cette interface, nous devons développer une méthode `marshall()`. De façon similaire, une classe Java destinée à la désérialisation d'objets doit implémenter l'interface `org.apache.soap.util.xml.Deserializer` et fournir une méthode `unmarshall()`. Il est parfaitement raisonnable de créer une classe prenant en charge les deux fonctions et nous allons donc adopter cette approche. Appelons notre classe de sérialisation personnalisée `SerialiseurMessageSourceDeDonnees`.

```
package javasoap.livre.ch6;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.util.*;
import org.apache.soap.rpc.SOAPContext;
import org.w3c.dom.*;
import java.io.*;
import java.util.*;
public class SerialiseurMessageSourceDeDonnees
                implements Serializer, Deserializer {

  public SerialiseurMessageSourceDeDonnees() {
  }
  public void marshall(String styleEncodage, Class typeJava,
               Object src, Object contexte, Writer receptacle,
               NSStack pileEN, XMLJavaMappingRegistry xjmr,
```

```
                SOAPContext ctx)
            throws IllegalArgumentException, IOException  {

        if(!typeJava.equals(MessageSourceDeDonnees.class)) {
          throw new IllegalArgumentException(
            "Impossible de sérialiser des instances autres que
                 javasoap.livre.ch6.MessageSourceDeDonnees");
        }

        pileEN.pushScope();

        if (src != null)
        {
          SoapEncUtils.generateStructureHeader(styleEncodage,
                                               typeJava, contexte,
                                               receptacle, pileEN, xjmr);
          MessageSourceDeDonnees msg = (MessageSourceDeDonnees)src;
          String donnees = msg.toString();
          receptacle.write(donnees);
          receptacle.write("</" + contexte + '>');
        }
        else
        {
          SoapEncUtils.generateNullStructure(styleEncodage,
                                               typeJava, contexte,
                                               receptacle, pileEN, xjmr);
        }
        pileEN.popScope();
      }
      public Bean unmarshall(String styleEncodage, QName
                  typeElement, Node src,
                  XMLJavaMappingRegistry xjmr, SOAPContext ctx)
              throws IllegalArgumentException {

        Element elem = (Element)src;
        String value = DOMUtils.getChildCharacterData(elem);

        MessageSourceDeDonnees msg = null;
        if(value!=null && !((value=value.trim()).equals(""))) {
           msg = new MessageSourceDeDonnees();
           msg.analyseMessageFormate(value);
        }
        return new Bean(MessageSourceDeDonnees.class, msg);
      }
    }
```

Examinons tout d'abord la méthode `marshall()` qui sérialise une instance de `MessageSourceDeDonnees`. Elle commence par vérifier que la valeur du paramètre `javaType` est bien `MessageSourceDeDonnees.class` car ce sérialiseur personnalisé ne peut traiter d'instance d'un autre type d'objet. Si la classe est d'un type incorrect, `marshall()` ren-

voie une `IllegalArgumentException` accompagnée d'une description appropriée. (Nous n'avons pas encore parlé des exceptions et vous vous demandez donc peut-être quelle est la conséquence de cette action. Patientez jusqu'au chapitre 7 qui traite de ce sujet dans le détail.) Nous appelons ensuite `pileEN.pushScope()`. La variable `pileEN` passée en paramètre, représente l'espace de noms hiérarchique du message SOAP que nous sérialisons. Cet objet agit comme un pile de définitions d'espaces de noms. À chaque fois que la méthode `marshall()` du sérialiseur est appelée, nous étendons la portée de l'espace de noms. Cette technique nous permet d'effectuer les déclarations d'espaces de noms à notre niveau de la hiérarchie. Avant de sortir de la méthode `marshall()`, nous devons nous assurer que nous avons vidé la pile. Sans cela, nous courrons le risque d'écraser les déclarations d'espaces de noms des éléments XML qui nous précèdent dans la hiérarchie.

Une fois la portée de l'espace de noms étendue, nous devons nous assurer que ce que nous avons à sérialiser n'est pas nul. Si `src` est nul, nous appelons la méthode `SoapEncUtils.generateNullStructure()`, qui prendra en charge pour nous la création d'un élément nul. Si `src` n'est pas nul, (le cas le plus intéressant), nous appelons `SoapEncUtils.generateStructureHeader()` pour la première étape de la création du message SOAP. Cette méthode crée la balise de l'élément et inscrit les attributs et les valeurs associées nécessaires. Nous n'avons pas à nous en préoccuper. Notre tâche consiste à traiter la sérialisation des données et non les détails des balises XML. Enfin presque. Malgré tout, nous aurons à faire un travail de nettoyage à la fin, comme vous le verrez sous peu.

Comme nous savons que `src` est d'un type valide, nous le transtypons en `MessageSourceDeDonnees` et le référençons dans la variable `msg`. Tout ce que nous avons à faire maintenant, est d'appeler la méthode `msg.toString()`, qui renvoie les données du message source dans le format requis. Le message correctement mis en forme est maintenant stocké dans la variable `donnees`. La méthode `marshall()` reçoit un paramètre nommé `receptacle`, qui est une instance de `java.io.Writer`. C'est là que s'effectuera toute la sérialisation de l'enveloppe SOAP. Pour le moment, `receptacle` est positionné à l'endroit où la sérialisation réelle des données s'effectue. L'appel de la méthode `receptacle.write()` à l'aide du paramètre `donnees` écrit le message source dans le `receptacle`. La dernière étape consiste à écrire la balise de fin pour cet élément, même si l'écriture de la balise de l'élément a été prise en charge pour nous. Nous avons besoin du nom de l'élément pour produire sa balise fermante. Ce nom est disponible dans l'objet `contexte` passé comme l'un des paramètres à `marshall()`. Nous appelons `receptacle.write()` encore une fois pour fermer cet élément. Cela fonctionne car `contexte` implémente la méthode `toString()`, qui renvoie le nom de l'élément en cours de sérialisation. C'est tout. Récupérez l'espace de noms en appelant `pileEN.popStack()` et nous en aurons fini.

Examinons maintenant le processus de désérialisation : transformer les données encodées en une instance de classe Java. Ce processus est pris en charge par la méthode `unmarshall()` du sérialiseur. La désérialisation requiert l'utilisation de certaines API de traitement XML fournies par Apache SOAP. Nous transtypons le paramètre `src` en une instance de `org.w3c.dom.Element`, qui est une classe utilisée pour représenter un élément XML. Nous allons ensuite récupérer les données de l'élément mises en forme. Ceci constituera le message source dans le format décrit au début de cette section. On parvient à cela en passant la variable `elem` à la méthode `getChildCharacterData()` de la classe `org.apache.soap.util.xml.DOMUtils`. La valeur chaîne renvoyée lors de l'ap-

pel et sauvegardée dans value, contient le message source mis en forme. Il est naturellement recommandé de vérifier que le message contient bien des données et c'est pourquoi nous vérifions préalablement qu'il n'est pas nul ou constitué d'une chaîne vide. S'il contient des données, une nouvelle instance de `MessageSourceDeDonnees` est créée et les données sont envoyées à sa méthode `analyseMessageFormate()`. À ce stade, nous avons réussi à désérialiser les données et les avons utilisées pour renseigner une instance de `MessageSourceDeDonnees`. La méthode `unmarshall()` doit renvoyer une instance de `org.apache.soap.util.Bean`.[1] Cette classe est utilisée pour associer un objet Java à sa classe. Vous vous demandez peut-être à quoi cela peut-il bien servir puisque l'on peut déterminer la classe d'un objet en invoquant sa méthode `getClass()`. Appeler `getClass()` pose un problème si l'objet a été implémenté à l'aide d'une sous-classe de la classe correspondante et c'est tout bonnement impossible si l'objet est `null`. La dernière étape sera donc de créer une instance de `org.apache.soap.util.Bean`, à l'aide de `MessageSourceDeDonnees.class` en premier paramètre du constructeur et `msg` en second.

Nous avons maintenant implémenté les fonctionnalités de sérialisation et désérialisation de notre sérialiseur personnalisé. Nous pouvons continuer et déployer `urn:ServiceSourceDeDonnees`. Ce déploiement nécessite d'inclure une entrée dans le descripteur de déploiement, qui mettra en correspondance notre type personnalisé avec la bonne classe Java et l'associera au sérialiseur personnalisé que nous venons de créer. Voici le descripteur :

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:ServiceSourceDeDonnees">
  <isd:provider
     type="java"
     scope="Application"
     methods="envoieMessage">
    <isd:java
       class="javasoap.livre.ch6.ServiceSourceDeDonnees"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:x="urn:ServiceSourceDeDonnees" qname="x:MessageSourceDeDonnees"
      javaType="javasoap.livre.ch6.MessageSourceDeDonnees"
       java2XMLClassName="javasoap.livre.ch6.SerialiseurMessageSourceDeDonnees"
       xml2JavaClassName="javasoap.livre.ch6.SerialiseurMessageSourceDeDonnees"/>
  </isd:mappings>
</isd:service>
```

1. Ne soyez pas troublés par le fait que cette classe se nomme `Bean`. Elle n'est pas spécialement liée à la classe `BeanSerializer` avec laquelle nous avons travaillé au chapitre précédent. Il s'agit juste d'un nom provenant d'un autre projet.

La section des correspondances contient une entrée pour MessageSourceDeDonnees, qualifiée par un espace de noms qui utilise le nom du service lui-même. Nous assignons la valeur javasoap.livre.ch6.SerialiseurMessageSourceDeDonnees aux attributs java2XMLClassName et xml2JavaClassName, ce qui informe le système de l'existence de notre classe sérialiseur personnalisé. Le service peut maintenant être déployé.

Voici l'application cliente qui accèdera au service. Le processus est similaire à celui que nous avons utilisé au chapitre précédent : nous mettons nos types personnalisés en correspondance en tant que beans et utilisons SerialiseurMessageSourceDeDonnees pour sérialiser et désérialiser l'objet.

```
package javasoap.livre.ch6;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
public class ClientSourceDeDonnees {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");

      Call appel = new Call();

      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      appel.setTargetObjectURI("urn:ServiceSourceDeDonnees");
      appel.setMethodName("envoieMessage");
      appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      appel.setSOAPMappingRegistry(smr);
      SerialiseurMessageSourceDeDonnees msgSer =
                  new SerialiseurMessageSourceDeDonnees();
      // Mise en correspondance du type du message source
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
            new QName("urn:ServiceSourceDeDonnees", "MessageSourceDeDonnees"),
            MessageSourceDeDonnees.class, msgSer, msgSer);
      MessageSourceDeDonnees msg = new MessageSourceDeDonnees();
      msg.ajouteChamp("SYMBOLE", "XYZ");
      msg.ajouteChamp("DEMANDE", "PRIX");
      Vector params = new Vector();
      params.addElement(new Parameter("msg",
                  MessageSourceDeDonnees.class, msg, null));
      appel.setParams(params);

      try {
         Response rep = appel.invoke(url, "");
         Parameter ret = rep.getReturnValue();
```

```
            Object valeur = ret.getValue();
            System.out.println(valeur);
        }
        catch (SOAPException e) {
            System.err.println("SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

Nous instancions `SerialiseurMessageSourceDeDonnees` dans la variable nommée `msgSer`. Lors de l'appel de `smr.mapTypes()` pour mettre en correspondance notre type personnalisé et la classe `MessageSourceDeDonnees`, nous envoyons `msgSer` à la fois comme sérialiseur et désérialiseur. Ainsi, à l'invocation de la méthode de service `envoieMessage`, `SerialiseurMessageSourceDeDonnees` est utilisé pour sérialiser le message source et également pour désérialiser la valeur de retour de la méthode de service. Une instance de `MessageSourceDeDonnees` est créée et les champs SYMBOLE et DEMANDE sont ajoutés, contenant respectivement les valeurs XYZ et PRIX. Nous positionnons ensuite les paramètres de la méthode de service et l'invoquons. La valeur de retour d'`envoieMessage` est également un message source et elle est donc désérialisée dans une instance de `MessageSourceDeDonnees`. Nous n'avons pas à nous préoccuper de transtyper ce qui est renvoyé par `ret.getValue()`, car nous pouvons tirer parti de la méthode `toString()` pour afficher son contenu. L'exécution de cet exemple doit produire le résultat suivant :

```
2:SYMBOLE:XYZ:PRIX:123.5:
```

Ce sont les données mises en forme contenues dans l'instance de `MessageSourceDeDonnees` renvoyée. Le serveur y indique le symbole envoyé ainsi qu'un champ prix contenant la valeur 123.5. Examinons l'enveloppe SOAP envoyée au serveur :

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:envoieMessage
        xmlns:ns1="urn:ServiceSourceDeDonnees"
        SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">
       <msg
       xsi:type="ns1:MessageSourceDeDonnees">2:DEMANDE:PRIX:SYMBOLE:XYZ:
       </msg>
     </ns1:envoieMessage>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

L'élément `msg` est de type `MessageSourceDeDonnees`, qualifié par l'espace de noms `urn:ServiceSourceDeDonnees`. Vous pouvez voir que les données sont mises en forme selon nos spécifications. Voyons maintenant l'enveloppe SOAP renvoyée par la métho-

de de service, qui type également les données renvoyées en MessageSourceDeDonnees à l'aide de l'espace de noms urn:ServiceSourceDeDonnees. Nous sommes ainsi en mesure d'utiliser nos classes de sérialisation personnalisées pour prendre en charge le marshall et le démarshall de notre message source, aussi bien côté serveur que client.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:sendMessageResponse
       xmlns:ns1="urn:ServiceSourceDeDonnees"
       SOAP-ENV:encodingStyle=
         "http://schemas.xmlsoap.org/soap/encoding/">
         <return
         xsi:type="ns1:MessageSourceDeDonnees">2:PRIX:123.5:SYMBOLE:XYZ:
         </return>
    </ns1:sendMessageResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Sérialiser des tableaux épars

Nous avons parlé des tableaux épars au chapitre 3. Toutefois, ni Apache SOAP ni GLUE ne les supportent. Nous allons donc le faire nous-même.

L'écriture d'un sérialiseur de tableau épars à partir de rien peut représenter un travail considérable et nous allons donc commencer par récupérer le code source de org.apache.soap.encoding.soapenc.ArraySerializer dans notre distribution Apache SOAP. Nous allons modifier le code de ArraySerializer afin de créer une nouvelle classe nommée javasoap.livre.ch6.SerialiseurTableauEpars. Voici le code de la classe toute entière, mais ne soyez pas affolé. L'essentiel provient de la classe ArraySerializer. Nous allons seulement parcourir les modifications nécessaires à la prise en charge des tableaux épars.

```
package javasoap.livre.ch6;
import org.apache.soap.encoding.soapenc.*;
import java.beans.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import org.w3c.dom.*;
import org.apache.soap.util.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class SerialiseurTableauEpars implements Serializer, Deserializer
{
   public SerialiseurTableauEpars() {
   }
   public void marshall(String styleEncodage,
               Class typeJava, Object src,
```

```
        Object contexte, Writer receptacle, NSStack pileEN,
        XMLJavaMappingRegistry xjmr, SOAPContext ctx)
 throws IllegalArgumentException, IOException {
pileEN.pushScope();
String longueurChaine = src != null
                ? Array.getLength(src) + ""
                : "";
Class typeComposant = typeJava.getComponentType();
QName typeElement = xjmr.queryElementType(typeComposant,
                        Constants.NS_URI_SOAP_ENC);
if (src == null) {

   SoapEncUtils.generateNullArray(styleEncodage,
                                typeJava,
                                contexte,
                                receptacle,
                                pileEN,
                                xjmr,
                                typeElement,
                                longueurChaine);
}
else {

   SoapEncUtils.generateArrayHeader(styleEncodage,
                                typeJava,
                                contexte,
                                receptacle,
                                pileEN,
                                xjmr,
                                typeElement,
                                longueurChaine);
  receptacle.write(StringUtils.lineSeparator);
  int longueur = Array.getLength(src);
  for (int i = 0; i < longueur; i++) {

     pileEN.pushScope();
     Object valeur = Array.get(src, i);
     // nous ne sérialisons que si
     // l'élément existe, dans la mesure où
     // ceci est un sérialiseur de tableau épars
     if (valeur != null) {
      Class typeComposantReel = valeur.getClass();
       // utilisation d'un receptacle temporaire afin de pouvoir le
       // modifier avant d'en afficher le contenu
       StringWriter sw = new StringWriter();
       xjmr.marshall(styleEncodage,
               typeComposantReel, valeur, "item",
          sw, pileEN, ctx);

      receptacle.write("<item ");
```

```
            receptacle.write(pileEN.getPrefixFromURI(
                Constants.NS_URI_SOAP_ENC) +
               ":position=\"[" + i + "]\" ");
            receptacle.write(sw.toString().substring(6));
            receptacle.write(StringUtils.lineSeparator);
          }
          pileEN.popScope();
        }
        receptacle.write("</" + contexte + '>');
      }
      pileEN.popScope();
   }
   public Bean unmarshall(String styleEncodage,
                  QName typeElement, Node src,
                  XMLJavaMappingRegistry xjmr, SOAPContext ctx)
             throws IllegalArgumentException {

      Element racine = (Element)src;
      String nom = racine.getTagName();
      QName typeItemTableau = new QName("", "");
      Object tableau = getNewArray(styleEncodage, racine,
                           typeItemTableau, xjmr);
      if (SoapEncUtils.isNull(racine)) {
         return new Bean(tableau.getClass(), null);
      }
      Element tempEl = DOMUtils.getFirstChildElement(racine);
      while (tempEl != null) {
         String declEncStyle = DOMUtils.getAttributeNS(tempEl,
                Constants.NS_URI_SOAP_ENV,
                Constants.ATTR_ENCODING_STYLE);
         String styleEncReel = declEncStyle != null
                              ? declEncStyle
                              : styleEncodage;
         QName declItemType =
                SoapEncUtils.getAttributeValue(tempEl,
                Constants.NS_URI_CURRENT_SCHEMA_XSI,
                Constants.ATTR_TYPE,
                "tableau item", false);
         QName actualItemType = declItemType != null
                             ? declItemType
                             : typeItemTableau;
         Bean itemBean = xjmr.unmarshall(styleEncReel,
                             actualItemType,
                             tempEl, ctx);
         // récupère la position dans le tableau
         String pos = DOMUtils.getAttributeNS(tempEl,
                              styleEncodage, "position");

         int right = pos.indexOf(']');
         String substr = pos.substring(1, right);
```

```
            int idx = Integer.parseInt(substr);
            Array.set(tableau, idx, itemBean.value);
            tempEl = DOMUtils.getNextSiblingElement(tempEl);
        }
        return new Bean(tableau.getClass(), tableau);
    }
    public static Object getNewArray(String styleEncodage,
                    Element arrayEl,
                    QName typeItemTableau,
                    XMLJavaMappingRegistry xjmr)
             throws IllegalArgumentException {
        QName arrayTypeValue =
               SoapEncUtils.getAttributeValue(arrayEl,
               Constants.NS_URI_SOAP_ENC,
               Constants.ATTR_ARRAY_TYPE,
               "tableau", true);
        String arrayTypeValueNamespaceURI =
               arrayTypeValue.getNamespaceURI();
        String arrayTypeValueLocalPart =
               arrayTypeValue.getLocalPart();
        int leftBracketIndex =
               arrayTypeValueLocalPart.lastIndexOf('[');
        int rightBracketIndex =
               arrayTypeValueLocalPart.lastIndexOf(']');
        if (leftBracketIndex == -1
          || rightBracketIndex == -1
          || rightBracketIndex < leftBracketIndex) {
            throw new IllegalArgumentException(
                    "Malformed arrayTypeValue '" +
                     arrayTypeValue + "'.");
        }
        String componentTypeName =
              arrayTypeValueLocalPart.substring(0,
              leftBracketIndex);
        if (componentTypeName.endsWith("]")) {
           throw new IllegalArgumentException(
              "Les tableaux de tableaux ne sont pas " +
              "supportés '" + arrayTypeValue + "'.");
        }
        typeItemTableau.setNamespaceURI(arrayTypeValueNamespaceURI);
        typeItemTableau.setLocalPart(componentTypeName);
        int longueur = DOMUtils.countKids(arrayEl,
            Node.ELEMENT_NODE);
        String longueurChaine =
            arrayTypeValueLocalPart.substring(leftBracketIndex + 1,
                                              rightBracketIndex);
        if (longueurChaine.length() > 0) {
           if (longueurChaine.indexOf(',') != -1) {
              throw new IllegalArgumentException(
                   "Les tableaux multi-dimensionnels " +
```

```
                "ne sont pas supportés '" + longueurChaine + "'.");
            }
            try {
               int explicitLength = Integer.parseInt(longueurChaine);
               longueur = explicitLength;
            }
            catch (NumberFormatException e) {
               throw new IllegalArgumentException(
                     "La longueur du tableau n'est pas un " +
                     "entier valide '" + longueurChaine + "'.");
            }
         }
         Class typeComposant = xjmr.queryJavaType(typeItemTableau,
                  styleEncodage);
         return Array.newInstance(typeComposant, longueur);
      }
   }
```

Voyons tout d'abord le code de la méthode `marshall()`. Cette méthode est appelée lorsque le tableau est sérialisé. La sérialisation d'un tableau épars ne requiert pas un traitement particulier de ses éléments. Il n'y a pas d'attributs additionnels et les valeurs d'attributs employées sont identiques à celles utilisées pour un tableau sérialisé normalement. La seule modification consiste dans le fait que chaque élément du tableau requiert un attribut `position` qui indique où l'élément est situé dans le tableau. L'élément position doit être qualifié par un espace de noms à l'aide de `http://schemas.xmlsoap.org/soap/encoding/`. Il faut également que le sérialiseur ignore tout élément `null` du tableau plutôt que de l'inclure dans la sérialisation.

Examinez le code de la méthode `marshall()` à l'endroit où la longueur du tableau est déterminée par l'appel à `Array.getLength()`. La valeur est renvoyée dans la variable entière `length`, utilisée dans la boucle `for` suivante. Dans cette boucle, tous les éléments du tableau sont traités.[2] Nous obtenons une référence à chaque élément sous forme d'un `Object` nommé `valeur` et vérifions si l'objet est `null`. Si c'est le cas, nous l'ignorons et poursuivons avec l'élément suivant. S'il n'est pas `null`, nous commençons par déterminer la classe runtime de l'objet `valeur`. Après cela, nous devrions commencer à écrire la balise de l'élément du tableau, mais Apache SOAP ne fournit aucun mécanisme de sérialisation des attributs d'un élément. Ceci est un problème dans la mesure où nous devons inclure un attribut `position` pour chaque élément du tableau épars. Nous voulons cependant continuer à tirer parti du code Apache SOAP prenant en charge l'écriture des éléments et de leurs valeurs.

Occupons-nous d'abord de cette partie. Nous créons une instance de `java.io.StringWriter` nommée `sw`. Nous devrions passer normalement le `receptacle` à la méthode `xjmr.marshall()`, qui sérialise l'élément et l'écrit dans le `receptacle`. Au lieu de passer le `receptacle`, nous passons l'instance `StringWriter`, `sw`. Ceci a pour effet de stocker la forme sérialisée de l'élément du tableau que nous utiliserons sous peu. Le quatrième pa-

2. En pratique, ce n'est probablement pas la meilleure façon de représenter un tableau épars en Java, car cela requiert de boucler dans tout le tableau. Les très gros tableaux épars contenant peu de valeur devraient être implémentés par une représentation plus intelligente.

ramètre passé à `xjmr.marshall()` est `"item"`, qui est le nom de balise pour l'élément. Dans la mesure où c'est cela que nous voulons écrire ensuite dans le `receptacle`, nous le faisons nous-même en appelant `receptacle.write` avec le paramètre `"item "`. Nous avons là l'opportunité de renseigner une paire attribut/valeur dans le `receptacle`, car nous sommes positionné exactement là où nous le souhaitions, après l'écriture de la balise de l'élément. Nous devons qualifier l'attribut par un espace de noms et utilisons donc `nsStatck.getPrefixFromURI()` pour retrouver l'identifiant utilisé pour l'espace de noms d'encodage SOAP. La constante `Constants.NS_URI_SOAP_ENC` représente cet espace de noms et est donc passée en paramètre à `getPrefixFromURI()`. Nous commençons à construire la paire complète attribut/valeur pour l'attribut `position` à l'aide de l'identifiant d'espace de noms récupéré. Nous savons que le nom de l'attribut est `position` et que l'indice de l'élément du tableau correspond à la valeur du compteur de boucle `i`. Tout ceci est mis en forme en concaténant ces valeurs et en envoyant le résultat à `receptacle.write()`. Comme la valeur de l'attribut `position` est entourée de guillemets, nous utilisons une séquence d'échappement pour inclure ces guillemets dans la chaîne.

Nous voulons ensuite transférer les données sérialisées du `StringWriter` temporaire vers le `receptacle`. Mais nous ne voulons pas de balise d'élément au début car nous avons déjà écrit la balise et ses attributs dans le `receptacle`. Comme nous avons utilisé le même nom de balise, `item`, nous pouvons ignorer les six premiers caractères des données situées dans l'instance du `StringWriter` et récupérer le reste. (Les six caractères que nous souhaitons ignorer sont `<`, `item` et l'espace qui le suit.) Nous appelons donc `sw.toString().substring(6)` et envoyons le résultat à `receptacle.write()`. Nous avons ainsi réussi à inscrire l'attribut dans l'élément tout en continuant à tirer parti de l'API Apache SOAP pour effectuer l'essentiel du travail. Le reste du code de `marshall()` ajoute un saut de ligne et la balise fermante au `receptacle` et termine ainsi la sérialisation de l'élément du tableau.

Voyons ensuite la méthode `unmarshall()` utilisée pour désérialiser un tableau épars. Pour cela, nous n'avons pas grand-chose à faire. Presque tous le code est consacré à déterminer la taille et le type du tableau et à allouer une instance de tableau appropriée. Nous avons juste à déterminer la valeur de l'attribut `position` afin de pouvoir insérer l'instance de l'élément désérialisé au bon endroit dans le tableau. Nous pouvons récupérer la valeur de l'attribut en appelant `DOMUtils.getAttributeNS`. Nous devons passer à la méthode l'élément du document XML, le style d'encodage utilisé et le nom de l'attribut. L'élément se trouve dans la variable `tempEl`, qui a été auparavant traitée par la méthode. Le style d'encodage se trouve dans le paramètre `styleEncodage` qui a été envoyé à la méthode `unmarshall()`. Et nous savons naturellement que le nom de l'attribut est `position`. La valeur de l'attribut sera un entier entouré de crochets. Nous devons donc l'en extraire et l'envoyer à `Integer.parseInt()` de façon à récupérer une valeur `int` représentant la position du tableau. Cette valeur est envoyée en second paramètre à la méthode `Array.set()`, qui place l'élément dans le tableau à la position spécifiée. Tout le reste est déjà fait.

Maintenant que nous avons notre sérialiseur de tableaux épars, nous devons trouver comment l'utiliser. Après tout, Apache SOAP, par défaut, utilisera la classe `ArraySerializer` pour traiter la sérialisation et la désérialisation des tableaux. Nous devons donc trouver un moyen de supplanter ce comportement. Ma première approche serait de modifier la classe `org.apache.soap.encoding.SOAPMappingRegistry` fournie par Apa-

che SOAP pour la forcer à utiliser SerialiseurTableauEpars au lieu de ArraySerializer. Le problème serait toutefois que cela rendrait invalide tout tableau sérialisé sans paramètres de position. Mauvaise idée... Nous aurions pu rendre le SerialiseurTableauEpars suffisamment souple pour traiter toutes les possibilités mais cela l'aurait rendu très complexe et aurait alourdi le code. Si vous voulez vous lancer dans cette voie, je suis prêt à parier que les développeurs d'Apache seront prêts à inclure vos modifications dans leur code. Une approche plus simple serait d'inclure le sérialiseur de tableaux épars dans la section de mise en correspondance des services qui utilisent ce type de tableaux. Cela signifie que nous n'aurons pas à modifier le code source Apache SOAP, ce qui aurait été fort gênant si des modifications dans les versions à venir d'Apache avaient été incompatibles avec notre version modifiée. Nous allons donc utiliser l'approche par les services.

Ajoutons une méthode nommée echoActions au service urn:ServiceSourceDeDonnees. Cette méthode reçoit un tableau de chaînes de caractères en paramètre et utilise ce même tableau comme valeur de retour. Voici la nouvelle version de la classe ServiceSourceDeDonnees comprenant l'ajout de echoActions() :

```
package javasoap.livre.ch6;
public class ServiceSourceDeDonnees {
   public ServiceSourceDeDonnees() {
   }
   public String[] echoActions(String[] actions) {
      return actions;
   }
   public MessageSourceDeDonnees envoieMessage(MessageSourceDeDonnees msg) {
      // nous pouvons retirer le message mis en forme ici
      // en appelant msg.toString();
      String action = msg.getValeurChamp("SYMBOLE");
      String requete = msg.getValeurChamp("DEMANDE");
      MessageSourceDeDonnees res = new MessageSourceDeDonnees();
      res.ajouteChamp("SYMBOLE", action);
      if (requete.equals("PRIX"))
         res.ajouteChamp("PRIX", "123.5");
      return res;
   }
}
```

Nous devons ajouter une entrée de mise en correspondance au descripteur de déploiement de façon à ce que le sérialiseur de tableaux épars soit utilisé. Le type que nous voulons mettre en correspondance est Array, qualifié par l'espace de noms http://schemas.xmlsoap.org/soap/encoding/. Ceci correspond au tableau SOAP standard utilisant l'encodage SOAP standard. Cette entrée supplante les classes par défaut de sérialisation et désérialisation de tableaux. Nous savons déjà que la classe SerialiseurTableauEpars sera utilisée comme valeur des attributs java2XMLClassName et xml2JavaClassName. Cela indique au système d'utiliser notre nouveau sérialiseur pour les tableaux. Nous devons maintenant spécifier la classe Java utilisée pour instancier un tableau Java. Cette classe provient de l'API de réflexion Java et est nommée java.lang.reflect.Array. Nous allons utiliser cette valeur pour l'attribut javaType de l'entrée de mise en correspondance. Le nom de méthode echoActions est ajouté à l'attribut methods de la balise service.

Nous avons toutefois encore un problème à résoudre. Le code Java Apache SOAP ne s'attend pas à ce que les sérialiseurs standard soient supplantés et fait certaines suppositions quant à ces séraliseurs et au registre de mise en correspondance. Même si nous avons spécifié `SerialiseurTableauEpars` pour `java2xmlClassName` dans l'entrée de mise en correspondance, il ne sera pas utilisé lorsque notre service tentera de renvoyer un tableau depuis `echoActions`. Le moyen de contournement est d'écrire notre propre registre de mise en correspondance, ce qui est plus simple que cela n'en a l'air. Voyons comment procéder. Le nouveau registre s'appellera `javasoap.livre.ch6.RegistreCorrespSOAPAmeliore`. Nous devront indiquer ce registre dans le descripteur de déploiement en incluant l'attribut `defaultRegistryClass` dans l'élément `isd:mappings`, en utilisant comme valeur le nom de la classe de notre nouveau registre. Voici le descripteur de déploiement modifié :

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:ServiceSourceDeDonnees">
  <isd:provider
     type="java"
     scope="Application"
     methods="envoieMessage echoActions">
    <isd:java
       class="javasoap.livre.ch6.ServiceSourceDeDonnees"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>

  <isd:mappings defaultRegistryClass=
                "javasoap.livre.ch6.RegistreCorrespSOAPAmeliore">
     <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="http://schemas.xmlsoap.org/soap/encoding/" qname="x:Array"
       javaType="java.lang.reflect.Array"
       java2XMLClassName="javasoap.livre.ch6.SerialiseurTableauEpars"
       xml2JavaClassName="javasoap.livre.ch6.SerialiseurTableauEpars"/>
     <isd:map
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:x="urn:ServiceSourceDeDonnees" qname="x:MessageSourceDeDonnees"
      javaType="javasoap.livre.ch6.MessageSourceDeDonnees"
      java2XMLClassName="javasoap.livre.ch6.SerialiseurMessageSourceDeDonnees"
      xml2JavaClassName="javasoap.livre.ch6.SerialiseurMessageSourceDeDonnees"/>
  </isd:mappings>
</isd:service>
```

Tout ce que notre nouveau registre doit faire est de renvoyer une instance de `SerialiseurTableauEpars` au lieu de `ArraySerializer` lorsqu'il est interrogé dans le cadre du traitement d'un sérialiseur de tableaux. La classe `RegistreCorrespSOAPAmeliore` étend la classe `SOAPMappingRegistry` fournie par Apache SOAP. Pour résoudre le problème, nous supplantons la méthode `querySerializer` de façon à ce qu'elle renvoie un `Seria-`

liseurTableauEpars lorsque nécessaire. La classe de base effectue l'essentiel du travail, mais lorsqu'elle essaie de renvoyer un ArraySerializer, nous renvoyons à la place un SerialiseurTableauEpars. Voici la nouvelle classe de registre :

```
package javasoap.livre.ch6;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.soapenc.*;
public class RegistreCorrespSOAPAmeliore
      extends org.apache.soap.encoding.SOAPMappingRegistry {
   SerialiseurTableauEpars serialEpars = new SerialiseurTableauEpars();
   public RegistreCorrespSOAPAmeliore() {
      super();
   }
   public Serializer querySerializer(Class javaType,
                String encodingStyleURI)
       throws IllegalArgumentException {

     Serializer s;
     try {
        s = super.querySerializer(javaType, encodingStyleURI);
        if (s instanceof ArraySerializer)
          return serialEpars;
     }
     catch (IllegalArgumentException e) {
        if (javaType != null
         && encodingStyleURI != null
         && encodingStyleURI.equals(Constants.NS_URI_SOAP_ENC)) {
            if (javaType.isArray()) {
              return serialEpars;
            }
        }
        throw e;
     }
     return s;
   }
}
```

Pourquoi avons-nous besoin d'une entrée de correspondance dans le descripteur de déploiement si nous utilisons l'attribut defaultRegistryClass de l'élément isd:service ? Parce que l'attribut par lui-même ne semble pas fonctionner mais la combinaison de l'attribut et de l'entrée de correspondance fonctionne. Ce moyen de contournement est assez simple et vous évite de plonger dans le code source Apache. Poursuivez et déployez le service.

Voyons maintenant ce qu'il y a à faire dans l'application cliente. Nous devons en réalité faire la même chose mais d'une manière différente. Nous spécifions les correspondances et utilisons la classe RegistreCorrespSOAPAmeliore au lieu de la classe SOAPMappingRegistry. Voici le code du client :

```
package javasoap.livre.ch6;
import java.net.*;
```

```
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
public class ClientDonneesSource2 {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");

      Call appel = new Call();

      RegistreCorrespSOAPAmeliore smr =
                      new RegistreCorrespSOAPAmeliore();
      appel.setTargetObjectURI("urn:ServiceSourceDeDonnees");
      appel.setMethodName("echoActions");
      appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      appel.setSOAPMappingRegistry(smr);
      SerialiseurTableauEpars serialTableau =
                      new SerialiseurTableauEpars();
      // Mise en correspondance du tableau
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
                new QName(Constants.NS_URI_SOAP_ENC, "Array"),
                java.lang.reflect.Array.class, serialTableau,
                serialTableau);
      String[] actions = new String[10];
      actions[2] = "XYZ";
      actions[5] = "ABC";
      Vector params = new Vector();
      params.addElement(new Parameter("msg", String[].class,
                              actions, null));
      appel.setParams(params);

      try {
         Response rep = appel.invoke(url, "");
         Parameter ret = rep.getReturnValue();
         String[] valeur = (String[])ret.getValue();
         int cnt = valeur.length;
         for (int i = 0; i < cnt; i++) {
            if (valeur[i] != null) {
               System.out.println("Element " + i + ": " + valeur[i]);
            }
         }
      }
      catch (SOAPException e) {
         System.err.println("SOAPException (" +
                         e.getFaultCode() + "): " +
```

```
                    e.getMessage());
            }
        }
    }
```

La variable `smr` est maintenant une instance de `RegistreCorrespSOAPAmeliore`, mais à partir de maintenant nous l'utiliserons comme dans les exemples précédents. Nous créons une instance de `SerialiseurTableauEpars` à utiliser dans le processus de mise en correspondance. Nous appelons ensuite `smr.mapTypes()`, en lui donnant le `SerialiseurTableauEpars` précédemment instancié comme objet à utiliser pour la sérialisation et la désérialisation et `java.lang.reflect.Array.class` comme classe à utiliser pour l'objet implémentant le tableau. Notez les paramètres utilisés pour le constructeur `Qname`. Le premier est l'espace de noms d'encodage SOAP standard et le second est `"Array"`. Ceci devrait supplanter la mise en correspondance par défaut.

Nous créons maintenant un tableau de type chaîne vide de taille 10 et assignons des valeurs aux positions 2 et 5. Les autres éléments du tableau restent nuls. N'assignez pas de chaînes vides car une chaîne vide n'est pas une valeur nulle. (Bien qu'utiliser une chaîne vide ne présente aucun risque. Vous récupèrerez simplement un élément sérialisé pour les 10 éléments du tableau.) La méthode de service `echoActions` est invoquée de la même façon que précédemment. La valeur de retour est transtypée en `String[]` afin que nous puissions itérer en elle à la recherche des valeurs. Si tout est correct, nous ne devrions trouver des objets qu'en positions 2 et 5 puisque le serveur renvoie le tableau que nous lui avons envoyé. Le résultat de l'exécution de cette application sera :

```
Element 2: XYZ
Element 5: ABC
```

Et voici l'enveloppe SOAP pour cet exemple. Cela ne montre rien d'autre que le fait que seuls les éléments non-nuls ont été sérialisés.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
 <ns1:echoActions
   xmlns:ns1="urn:ServiceSourceDeDonnees"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <msg
       xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
       xsi:type="ns2:Array" ns2:arrayType="xsd:string[10]">
         <item ns2:position="[2]" xsi:type="xsd:string">XYZ</item>
         <item ns2:position="[5]" xsi:type="xsd:string">ABC</item>
     </msg>
 </ns1:echoActions>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

L'élément `msg` est de type tableau de 10 chaînes et ne diffère en rien d'un tableau quelconque. Chaque élément `item` est explicitement typé en `xsd:string` et inclus un attribut `position` qui indique sa position dans le tableau. C'est exactement ce que nous attendions.

Sérialiser des collections

Les tableaux ne sont pas les seuls types de collections communément utilisés en programmation Java. Il serait également plaisant de pouvoir sérialiser des tables de hachage, des listes chaînées et d'autres types de collections. GLUE fournit une implémentation personnalisée pour les tables de hachage qui mérite que l'on s'y arrête. Cet exemple n'utilise qu'un seul des nombreux mécanismes fournis par GLUE pour cette sérialisation personnalisée. Elle nécessitera que vous écriviez le schéma du type personnalisé à la volée, ce qui bien entendu signifie que vous aurez besoin de comprendre les schémas. Ceci n'est pas le sujet de ce livre mais cela vaut néanmoins la peine de jeter un coup d'œil à cette partie du code.

Pour développer un sérialiseur personnalisé sous GLUE, nous allons partir d'une classe Java qui étend `electric.xml.io.Type`, la classe de base pour les types SOAP sous GLUE. La seule chose à faire est de supplanter les méthodes `writeSchema()`, `writeObject()` et `readObject()`. Voici l'implémentation GLUE de la classe `HashtableType` :

```
// copyright 2001 by The Mind Electric
package electric.xml.io.collections;
import java.io.*;
import java.util.*;
import electric.xml.*;
import electric.xml.io.*;
import electric.util.Value;
public class HashtableType extends Type
  {
  /**
   * @param schema
   */
  public void writeSchema( Element schema )
    {
    Element complexType = schema.addElement( "complexType" );
    complexType.setAttribute( "name", "Map" );
    Element sequence = complexType.addElement( "sequence" );
    Element element = sequence.addElement( "element" );
    element.setAttribute( "name", "item" );
    element.setAttribute( "minOccurs", "0" );
    element.setAttribute( "maxOccurs", "unbounded" );
    Element subType = element.addElement( "complexType" );
    Element subSequence = subType.addElement( "sequence" );
    Element key = subSequence.addElement( "element" );
    key.setAttribute( "name", "key" );
    key.setAttribute( "type", getName( key, Object.class ) );
    Element valeur = subSequence.addElement( "element" );
    valeur.setAttribute( "name", "valeur" );
    valeur.setAttribute( "type", getName( key, Object.class ) );
    }
  /**
   * @param writer
   * @param object
   * @throws IOException
```

```
 */
public void writeObject( IWriter writer, Object object )
  throws IOException
  {
  writer.writeType( this );
  Hashtable hashtable = (Hashtable) object;
  for( Enumeration enum = hashtable.keys();
                          enum.hasMoreElements(); )
    {
    Object key = enum.nextElement();
    Object valeur = hashtable.get( key );
    IWriter itemWriter = writer.writeElement( "item" );
    itemWriter.writeObject( "key", key );
    itemWriter.writeObject( "valeur", valeur );
    }
  }
/**
 * @param reader
 * @param value
 * @throws IOException
 */
public void readObject( IReader reader, Value value )
  throws IOException
  {
  Hashtable hashtable = new Hashtable();
  value.setObject( hashtable );
  IReader[] readers = reader.getReaders( "item" );
  for( int i = 0; i < readers.length; i++ )
    {
    IReader itemReader = readers[ i ];
    Object key = itemReader.readObject( "key" );
    Object theValue = itemReader.readObject( "value" );
    hashtable.put( key, theValue );
    }
  }
}
```

GLUE fournit un support natif de cette classe et nous n'aurons donc pas à l'enregistrer lorsque nous nous écrirons le code du service et de l'application cliente. Si vous souhaitez développer vos propres sérialiseurs de types personnalisés sous GLUE, vous devrez utiliser l'une des méthodes `mapClass()` de la classe `electric.xml.io.Mappings`. Ces méthodes indiquent à GLUE comment mettre en correspondance vos types personnalisés avec des sérialiseurs.

Le code de la méthode `writeSchema()` produit une définition de schéma pour la table de hachage. Au lieu d'examiner toutes les classes et méthodes utilisées pour effectuer cette tâche, voyons le schéma qui en résulte :

```
<complexType name="Map">
  <sequence>
    <element name="item" minOccurs="0" maxOccurs="unbounded">
      <complexType>
```

```
          <sequence>
            <element name="key" type="KEYTYPE"/>
            <element name="value" type="VALUETYPE"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
```

Ceci est la définition de schéma pour une table de hachage désignée également sous le nom de *map*. Souvenez-vous que chaque entrée d'une map contient une paire clé/valeur. Elle est définie comme étant un `complexType` nommé `Map` et contenant une séquence de 0 ou plus éléments. Chaque élément est lui-même un type complexe contenant une séquence de deux éléments, `key` et `value`. Ces deux éléments sont typés à l'aide des noms de types appropriés des éléments `key` et `value`. J'ai utilisé `KEYTYPE` et `VALUETYPE` comme paramètres fictifs. Dans le code, ces types sont déterminés à partir des classes des instances correspondantes d'objets Java. Nous verrons cela bientôt lorsque nous exécuterons un exemple utilisant ce sérialiseur personnalisé.

La méthode `writeObject()` sert à sérialiser le contenu du type personnalisé. Son premier paramètre est une instance de l'interface `electric.xml.io.IWriter`. Nous utiliserons cette interface pour écrire le contenu de l'objet à sérialiser. Le second paramètre est l'objet Java à sérialiser. La méthode commence par écrire le type de l'objet dans le flux, en envoyant `this` à la méthode `writeType()` du `writer`. Cela permet d'identifier l'objet à sérialiser comme étant de type `Map`. L'objet est ensuite transtypé en une instance de `Hashtable` afin que nous puissions accéder aux paires clé/valeur. Nous n'avons plus qu'à itérer dans les éléments de la table en utilisant l'`Enumeration` renvoyée par la méthode `hashtable.keys()`. Nous récupérons pour chaque élément les instances d'`Object` de la clé et de la valeur. Nous invoquons `writer.writeElement()` pour écrire chaque élément, ce qui renvoie une nouvelle instance de `IWriter` nommée `itemWriter`. Cette instance est utilisée pour renvoyer la clé et la valeur de cet élément en utilisant `itemWriter.writeObject()` qui recevra en paramètres le nom de l'élément et l'instance d'`Object` associée.

`readObject()` effectue la désérialisation. Son premier paramètre est une instance de l'interface `electric.xml.io.IReader`, utilisée pour lire le contenu à désérialiser. L'autre paramètre est une instance de `electric.util.Value`, contenant une instance de `java.lang.Object` et son type associé (une instance de `java.lang.Class`). La première étape sera de créer une nouvelle instance de `java.util.Hashtable` et de la passer en paramètre à `value.setObject()`. Nous appellerons ensuite `reader.getReaders()`, qui renverra un tableau d'instances `IReader`, une pour chaque paire clé/valeur. Pour chaque `IReader`, nous appellerons `getObject()` pour récupérer la clé et la valeur de l'élément. Ces deux objets sont ensuite ajoutés à la table de hachage à l'aide de la méthode `hashtable.put()`.

C'est tout ce qu'il y a à faire. Nous allons donc maintenant construire un service et une application cliente utilisant la table de hachage pour voir comment le sérialiseur personnalisé a été enregistré. Créons un nouveau service nommé `urn:AutreServiceSourceDonnees`. Celui-ci, tout comme le fournisseur de données précédent, contient une méthode nommée `envoieMessage()`. Mais cette fois, la méthode reçoit en paramètre une table de hachage dans laquelle les éléments contiennent les noms des champs et

leurs valeurs associées. Cela signifie que ce nouveau service doit mettre en forme les données comme spécifié au début de ce chapitre. Nous n'allons pas nous ennuyer avec cela dans la mesure où cela n'a pas de réel impact sur ce que nous sommes en train de faire. Nous allons nous concentrer sur la sérialisation personnalisée des tables de hachage. La classe `javasoap.livre.ch6.AutreServiceSourceDonnees` implémente le service.

```
package javasoap.livre.ch6;
import java.util.Hashtable;
public class AutreServiceSourceDonnees {
   public AutreServiceSourceDonnees() {
   }
   public Hashtable envoieMessage(Hashtable msg) {
      String action = (String)msg.get("SYMBOLE");
      String requete = (String)msg.get("DEMANDE");
      Hashtable res = new Hashtable();
      res.put("SYMBOLE", action);
      if (requete.equals("PRIX"))
         res.put("PRIX", "123.5");
      return res;
   }
}
```

La méthode `envoieMessage()` remplit globalement la même fonction que dans la classe `ServiceSourceDeDonnees`. La différence est que les messages sont passés à l'aide de tables de hachage plutôt que d'instances de `MessageSourceDeDonnees`. Pour déployer le service, nous allons utiliser la classe `AppServiceSourceDonnees` :

```
package javasoap.livre.ch6;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
import java.util.Hashtable;
public class AppServiceSourceDonnees {

   public static void main( String[] args )
                throws Exception {
      String ns = "urn:AutreServiceSourceDonnees";
      HTTP.startup("http://localhost:8004/glue");
      Context contexte = new Context();
      contexte.addProperty("activation", "application");
      contexte.addProperty("namespace", ns);
      Registry.publish(ns,
         javasoap.livre.ch6.AutreServiceSourceDonnees.class, contexte );
   }
}
```

Écrivons nous-mêmes l'interface pour la liaison au service, dans la mesure où nous allons tirer parti d'un type personnalisé déjà présent sous GLUE. Voici l'interface `javasoap.livre.ch6.IAutreServiceSourceDonnees`. Elle contient la méthode `envoieMessage()`, définie pour le service et qui reçoit une `Hashtable` en paramètre et en renvoie également une.

```
package javasoap.livre.ch6;
public interface IAutreServiceSourceDonnees {
  java.util.Hashtable envoieMessage(java.util.Hashtable msg);
}
```

ClientServiceSourceDonnees est une application qui se lie au service urn:AutreServiceSourceDonnees et passe une table de hachage à la méthode envoieMessage().

```
package javasoap.livre.ch6;

import electric.registry.RegistryException;
import electric.registry.Registry;
import java.util.*;

public class ClientServiceSourceDonnees {
   public static void main(String[] args) throws Exception
   {
      try {

        IAutreServiceSourceDonnees srv = (IAutreServiceSourceDonnees)
                                          Registry.bind(
         "http://localhost:8004/glue/urn:AutreServiceSourceDonnees.wsdl",
          IAutreServiceSourceDonnees.class);

        Hashtable msg = new Hashtable();
        msg.put("SYMBOLE", "XYZ");
        msg.put("DEMANDE", "PRIX");

        Hashtable resultat = srv.envoieMessage(msg);

        for (Enumeration e = resultat.keys(); e.hasMoreElements(); ) {
           String cle = (String)e.nextElement();
           String valeur = (String)resultat.get(cle);
           System.out.println(cle + ": " + valeur);
        }
    }
    catch (RegistryException e) {
       System.out.println(e);
    }
  }
}
```

La table de hachage renvoyée est énumérée et les paires clé/valeur sont affichées. Le résultat est:

```
SYMBOLE: XYZ
PRIX: 123.5
```

Examinons l'enveloppe SOAP envoyée au service lors de l'invocation de la méthode envoieMessage(). Cela nous permettra de voir la table de hachage sérialisée.

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
```

```
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
    xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
    soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
      <soap:Body>
        <n:sendMessageResponse xmlns:n='urn:AutreServiceSourceDonnees'>
          <Result href='#id0'/>
        </n:sendMessageResponse>
        <id0 id='id0' soapenc:root='0'
             xmlns:ns2='http://xml.apache.org/xml-soap'
             xsi:type='ns2:Map'>
          <item>
            <key xsi:type='xsd:string'>PRIX</key>
            <value xsi:type='xsd:string'>123.5</value>
          </item>
          <item>
            <key xsi:type='xsd:string'>SYMBOLE</key>
            <value xsi:type='xsd:string'>XYZ</value>
          </item>
        </id0>
      </soap:Body>
    </soap:Envelope>
```

Comme nous l'avons vu dans tous les autres exemples GLUE qui utilisaient des types complexes, les données réelles sont sérialisées séparément et le paramètre référence ces données. La table de hachage que nous avons envoyé se trouve dans l'élément `id0`. Elle est de type `Map`, qualifiée par l'espace de noms `http://xml.apache.org/xml-soap`. Cet espace de noms est utilisé car c'est l'Apache Group qui a créé le schéma pour encoder les tables de hachage. L'implémentation GLUE est en ce sens cohérente. Chaque élément enfant `item` contient des éléments `key` et `value` qui à leur tour contiennent les paires clé/valeur des entrées de la table. Dans ce cas, toutes les clés et les valeurs sont des chaînes car il s'agit du type utilisé dans l'exemple.

Dans ce chapitre, nous avons vu comment créer des sérialiseurs pour nos propres types personnalisés. La combinaison de ces techniques et du support natif de nombreux types, rend virtuellement sans limite les données qui peuvent être transmises par SOAP. Dans certains cas, vous vous rendrez compte que vous concevez vos classes avec SOAP à l'esprit ou au moins avec les possibilités et restrictions de l'implémentation choisie. Que la prise en compte des limites de votre implémentation ait du sens ou non dans la conception de votre projet, la sérialisation personnalisée ouvre de nombreuses possibilités lorsque vous avez besoin d'aller au-delà des possibilités de base.

7

Fautes et exceptions

Pour un logiciel, la différence entre bonne et piètre qualité tient souvent dans la façon dont sont gérés les erreurs et les problèmes. Il est plus aisé de traiter des processus où tout fonctionne correctement que de traiter des résultats anormaux. La programmation structurée ou orientée objet fournit de nombreuses techniques pour le traitement des erreurs. Les méthodes renvoient en général une valeur particulière comme `null` ou `-1`, pour indiquer une erreur quelconque. En Java, les méthodes lèvent souvent une exception indiquant que quelque chose d'anormal est survenu. Un bon logiciel est écrit de façon à prévoir certains types d'erreurs et est prévu pour que les actions appropriées soient effectuées dans ces cas. Dans la terminologie SOAP, ces circonstances inhabituelles ou exceptionnelles sont appelées des *fautes*. Les fautes surviennent toutes les fois qu'une méthode de service se révèle incapable de traiter ses paramètres en entrée et de renvoyer correctement un résultat. Un nombre considérable de raisons peut entraîner cela. Les causes les plus courantes des fautes sont l'envoi de valeurs de paramètres erronées, des problèmes sur le serveur de données ou des messages de requêtes SOAP de format incorrect. Je suis certain que vous pouvez imaginer sans peine de nombreuses autres causes et avez probablement eu à écrire du code permettant de les traiter. Nous verrons dans ce chapitre les mécanismes de génération et traitement des fautes sous SOAP.

Envoyer des exceptions côté serveur sous Apache SOAP

La façon la plus commune de générer une faute pour un service, est de renvoyer une exception. Apache SOAP possède un mécanisme de traitement des exceptions envoyées par les méthodes des classes Java implémentant les méthodes d'un service. L'information contenue dans l'exception est utilisée pour générer les fautes SOAP correspondantes qui seront envoyées plus tard à l'appelant.

Le contenu d'une faute SOAP a été traité plus haut au chapitre 2. Nous avions vu dans cette section que quatre codes de faute pouvaient être utilisés. L'une de ces possibilités est `SOAP-ENV:Server`, qui indique qu'un problème non lié au contenu de la requête est survenu durant le traitement du message. Ce code de faute peut être utilisé si, par exem-

ple, le serveur devant traiter le message n'est pas disponible. Créons un service très simple générant la plus élémentaire des fautes afin de voir le mécanisme de base utilisé sous Apache SOAP. Voici la classe `javasoap.livre.ch7.ServiceFauteExemple` qui implémente le service `urn:ServiceFauteExemple`:

```
package javasoap.livre.ch7;
import org.apache.soap.SOAPException;
public class ServiceFauteExemple {
   public ServiceFauteExemple() {
   }
   public int genereFaute()
        throws Exception {
      // quelque chose d'anormal est arrivé...
      throw new Exception();
   }
}
```

La déclaration de la méthode `genereFaute()` indique qu'une instance de `java.lang.Exception` peut être levée. En réalité, c'est même tout ce qu'elle peut faire. Nous simulons ce qui ce passe lorsqu'une méthode de service reçoit une demande qu'elle ne peut traiter en raison d'un problème survenu sur le serveur.

Voyons maintenant une application cliente appelant le service `genereFaute`. Jusqu'à présent, nous avons toujours obtenu les réponses d'un service web en appelant la méthode `getReturnValue()` de l'objet `Response`. Cette approche suppose toutefois que le service a été invoqué avec succès et ne tient pas compte du fait qu'une faute a pu être générée. Nous devons le demander à l'objet `Response` en appelant sa méthode `faultGenerated()`, qui renvoie la valeur `true` si c'est le cas et `false` sinon. Si une faute a été générée, nous pouvons procéder comme à notre habitude. Dans le cas contraire, nous pouvons identifier la faute à partir de l'objet `Response` en appelant sa méthode `getFault()`. Celle-ci renvoie une instance de `org.apache.soap.Fault`, une classe utilisée pour encapsuler le contenu d'une faute SOAP.

```
package javasoap.livre.ch7;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
public class ClientExemple {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");

      Call appel = new Call();

      appel.setTargetObjectURI("urn:ServiceFauteExemple");
      appel.setMethodName("genereFaute");
```

```
        try {
           Response rep = appel.invoke(url, "");
           if (rep.generatedFault()) {
              String code = rep.getFault().getFaultCode();
              String desc = rep.getFault().getFaultString();
              System.out.println(code + ": " + desc);         }
           else {
              Parameter ret = rep.getReturnValue();
              Object valeur = ret.getValue();
              System.out.println(valeur);
           }
        }
        catch (SOAPException e) {
           System.err.println("SOAPException (" +
                           e.getFaultCode() + "): " +
                           e.getMessage());
        }
     }
  }
```

En exécutant cet exemple, vous devriez obtenir le résultat suivant :

```
SOAP-ENV:Server: Exception from service object: null
```

Comme nous ne faisons rien de plus que renvoyer une instance de `java.lang.Exception` depuis notre méthode de service, le résultat est une faute SOAP de code `SOAP-ENV:Server`. La faute contient également une valeur par défaut pour la chaîne de caractères qui l'identifie ainsi que pour ses éléments acteurs, ainsi que pour sa source. Voici la faute SOAP renvoyée par la méthode de service `genereFaute` :

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>Exception from service object: null</faultstring>
<faultactor>/soap/servlet/rpcrouter</faultactor>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Mais où la faute a-t-elle donc été générée ? La réponse se trouve dans les descripteurs de déploiement que nous avons utilisé dans les exemples précédents. Chacun de ces descripteurs contient un élément `isd:faultListener` ressemblant à :

```
<isd:faultListener>org.apache.soap.server.DOMFaultListener
</isd:faultListener>
```

Cet élément spécifie la classe Java en charge du traitement des fautes dans votre service. Cette classe traite les exceptions renvoyées par votre service et les convertit en fautes SOAP. Malheureusement, l'information qui peut être extraite d'une instance de `java.lang.Exception` n'est pas si utile que cela. Apache SOAP contient une classe nom-

mée org.apache.soap.SOAPException qui étend java.lang.Exception et inclut des méthodes qui permettent de positionner ou récupérer un code de faute SOAP et son libellé. Cela est certainement plus utile et nous allons donc modifier la classe du service pour utiliser SOAPException :

```
package javasoap.livre.ch7;
import org.apache.soap.SOAPException;
public class ServiceFauteExemple {
   public ServiceFauteExemple() {
   }
   public int genereFaute()
        throws SOAPException {
      throw new SOAPException("SOAP-ENV:Server",
                          "Source de données indisponible ");
   }
}
```

Désormais, la méthode genereFaute() renvoie une instance de SOAPException. Considérons que la faute survient du fait de l'indisponibilité de la source de données utilisée par ce service. Il nous faut dans ce cas utiliser le code de faute SOAP-ENV:Server, qui est le premier paramètre du constructeur SOAPException. Le second paramètre décrit la faute. Ces deux valeurs sont utilisées pour renseigner les éléments faultcode et faultstring de la faute SOAP. Cette approche nous permet également de générer des fautes à l'aide du code de faute SOAP-ENV:Client, qui indique que la méthode de service a rencontré une requête incorrectement mise en forme ou quelque chose dans le corps SOAP qui empêche le traitement. Souvenez-vous des exemples de services fournisseurs d'informations boursières des chapitres précédents. Quelques-uns de ces exemples utilisaient des paires attributs/valeurs comme champs de message. Si un champ obligatoire venait à manquer, comme le champ SYMBOLE qui fournit le symbole de l'action, le serveur pourrait générer une faute en utilisant le code de faute SOAP-ENV:Client de la façon suivante :

```
throw new SOAPException("SOAP-ENV:Client", "SYMBOLE manquant");
```

C'est peut-être tout ce dont vous aurez besoin pour générer les fautes adéquates. Dans certains cas toutefois, vous pourrez renvoyer une information plus significative dans les fautes SOAP en utilisant l'élément detail. Cet élément n'est permis que lors de la description de fautes associées au corps du message de requête. Il n'a donc aucune utilité pour décrire par exemple une source de données indisponible, mais il peut sans aucun doute être utile en d'autres circonstances. (En réalité, je trouve cela réducteur car il existe de nombreuses situations dans lesquelles il serait utile de fournir les détails d'une faute qui n'est pas liée au corps de la requête. Supposons que, par exemple, la source de données soit indisponible mais qu'il en existe une autre susceptible d'accepter des requêtes. Vous pourriez vouloir que la faute contienne l'information nécessaire pour que le client utilise le service de remplacement. Naturellement, vous pourriez concevoir votre service pour qu'il utilise automatiquement des ressources de serveur alternatives, mais cela n'est pas toujours souhaitable.) Dans la section suivante, nous verrons comment fournir des détails de faute dans le contexte d'un code SOAP-ENV:Client.

Créer un module d'écoute de fautes sous Apache SOAP

Pour inclure un élément `detail` dans une faute, nous devons créer une nouvelle classe pour le module d'écoute en implémentant l'interface `org.apache.soap.server.SOAPFaultListener`. Cette interface définit une méthode nommée `fault()`, qui est appelée par Apache SOAP lorsqu'une faute doit être générée. Le seul paramètre à passer à cette méthode est une instance de `org.apache.soap.server.SOAPFaultEvent`,[1] qui transmet une instance de `org.apache.soap.Fault` et la `SOAPException` renvoyée par la méthode de service. L'objet faute contient des méthodes pour récupérer et positionner le code, le libellé et l'acteur de la faute ainsi que les détails. C'est cet objet que nous allons manipuler pour fournir l'information détaillée.

Mais avant de créer le module d'écoute de faute, nous avons besoin d'un mécanisme pour l'alimenter avec les données détaillées. Il n'y a pas de lien direct entre nos méthodes de service et notre module d'écoute, mais nous pouvons tirer parti du mécanisme des exceptions. Puisque nous avons vu que la `SOAPException` renvoyée par la méthode de service était accessible par le module d'écoute, nous pouvons hériter de `SOAPException` et créer une nouvelle classe correspondant à nos besoins. Mais avant cela, il nous faut une classe pour transmettre les informations détaillées. Utilisons une `java.util.Hashtable`. Nous pouvons de cette façon positionner un nombre quelconque de paires attribut/valeur pour représenter l'information détaillée sur la faute. Gardant tout cela à l'esprit, créons un nouveau type d'exception contenant une propriété nommée `Detail` qui sera une instance de `Hashtable`.

```
package javasoap.livre.ch7;
import org.apache.soap.*;
import java.util.Hashtable;
public class ExceptionSourceDonnees extends SOAPException {
   Hashtable _detail;
   public ExceptionSourceDonnees(String fcode, String fchaine) {
      super(fcode, fchaine);
   }
   public void setDetail(Hashtable detail) {
      _detail = detail;
   }
   public Hashtable getDetail() {
      return _detail;
   }
}
```

Le code est relativement simple. Dans la mesure où `ExceptionSourceDonnees` étend `SOAPException`, nous pouvons utiliser une instance de cette nouvelle classe à la place de l'instance de `SOAPException` utilisée précédemment. Le constructeur reçoit le code et le libellé de la faute en paramètres et les transmet au constructeur de la superclasse. Nous

1. Si êtes familiarisé avec les modèles JavaBeans pour les évènements, vous devriez reconnaître cette convention de nommage. L'évènement est une `SOAPFault`. L'interface du module d'écoute est nommée en concaténant `Listener` au nom de l'évènement et l'objet transportant l'information sur l'évènement est nommé par ajout de `Event`.

avons positionné et récupéré les accesseurs de la propriété _detail, définie comme une instance de Hashtable.

Nous pouvons maintenant modifier la classe de service pour renvoyer une instance de ExceptionSourceDonnees après l'avoir renseignée par quelque information détaillée. Voici le nouveau code de ServiceFauteExemple avec les modifications. Nous avons inclus dans les détails, les champs codeCause, descriptionCause, fournisseurSubstitution et contactSubstitution qui contiennent les valeurs appropriées. Ces détails, contenus dans une table de hachage, sont envoyés à l'instance de ExceptionSourceDonnees à l'aide de la méthode setDetail(). Nous pouvons lever une ExceptionSourceDonnees avec succès car elle est une sous-classe de SOAPException. Une autre solution serait de modifier la définition de la méthode pour spécifier qu'elle renvoie une ExceptionSourceDonnees.

```
package javasoap.livre.ch7;
import org.apache.soap.SOAPException;
import java.util.Hashtable;
public class ServiceFauteExemple {
   public ServiceFauteExemple() {
   }
   public int genereFaute()
        throws SOAPException {
      ExceptionSourceDonnees e = new ExceptionSourceDonnees("SOAP-ENV:Server",
                                 "Source de données indisponible");
      Hashtable detail = new Hashtable();
      detail.put("codeCause", "199");
      detail.put("descriptionCause", "Panne électrique");
      detail.put("fournisseurSubstitution", "www.mindstrm.com");
      detail.put("contactSubstitution", "rob@mindstrm.com");
      e.setDetail(detail);
      throw e;
   }
}
```

Nous pouvons maintenant nous pencher sur notre nouvelle classe de module d'écoute de faute. Appelons-la javasoap.livre.ch7.ModEcouteFauteSrcDonnees. Elle implémente l'interface org.apache.soap.server.SOAPFaultListener ainsi que la méthode requise fault().

```
package javasoap.livre.ch7;
import org.apache.soap.*;
import org.apache.soap.server.*;
import java.util.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import javax.xml.parsers.DocumentBuilder;

public class ModEcouteFauteSrcDonnees
       implements SOAPFaultListener {
   public ModEcouteFauteSrcDonnees() {
```

```
    }
    public void fault(SOAPFaultEvent evenement) {

       ExceptionSourceDonnees ex =
                        (ExceptionSourceDonnees)evenement.getSOAPException();
       Vector v = new Vector();
       DocumentBuilder constructeur =
                XMLParserUtils.getXMLDocBuilder();
       Document doc = constructeur.newDocument();
       Hashtable detail = ex.getDetail();
       Enumeration e = detail.keys();
       while (e.hasMoreElements()) {
          String nom = (String)e.nextElement();
          String valeur = (String)detail.get(nom);
          Element elem = doc.createElement(nom);
          Text txt = doc.createTextNode("");
          txt.setData(valeur);
          elem.appendChild(txt);
          v.addElement(elem);
       }

       evenement.getFault().setDetailEntries(v);
    }
}
```

Avant de décrire la logique en détail, voyons de façon générale ce que la méthode `fault()` a à faire. Elle reçoit un `SOAPFaultEvent` en réponse à l'`ExceptionSourceDonnees` renvoyée par la méthode de service. Cet évènement référence à la fois l' `ExceptionSourceDonnees` et la `Fault` réelle envoyée à l'appelant. Il nous faut donc extraire l'information détaillée située dans `ExceptionSourceDonnees`, la transformer en un groupe d'éléments XML pouvant être ajoutés à la `Fault`, puis inclure ces entrées dans la `Fault` de façon à ce qu'elles se retrouvent dans le document SOAP renvoyé à l'appelant.

La méthode `fault()` commence par récupérer l'`ExceptionSourceDonnees` renvoyée par la méthode de service. Pour cela, nous appliquons la méthode `getSOAPException()` à l'`evenement`. Le résultat est transtypé en une `ExceptionSourceDonnees`, car c'est là que nous trouverons les détails de la faute. Une instance de `java.util.Vector` est créée pour stocker les éléments de la section `detail` au fur et à mesure de leur création.

Il nous faut maintenant créer les éléments XML enfants de l'élément `detail`, en utilisant des API de manipulation de XML qui ne sont pas spécifiques à SOAP. Si vous souhaitez vous former à la manipulation XML en Java, référez-vous à l'ouvrage de Brett McLaughlin, *Java et XML* édité chez O'Reilly. Nous n'allons rien faire de compliqué cela dit; vous devriez suivre sans difficulté. Nous appelons `XMLParserUtils.getXMLDocBuilder` pour obtenir une instance de l'interface `DocumentBuilder`, utilisée pour créer un nouveau document. Ce `Document` est un document XML que nous utiliserons pour renseigner nos entrées détaillées.

Les détails sont obtenus à l'aide de la méthode `getDetail()` d'`ExceptionSourceDonnees`. L'étape suivante consiste à énumérer les entrées de la table de hachage pour en retirer les paires nom/valeur qui constituent les données détaillées. Pour chaque paire,

nous créons un nouvel élément XML en appelant `doc.createElement()` et en utilisant le nom comme nom d'élément. Nous créons alors un nœud texte et lui affectons la valeur récupérée dans la table de hachage. Le nœud texte est maintenant ajouté à l'élément et l'élément est ajouté au vecteur que nous avons créé plus tôt. Une fois cela fait pour chacune des paires, le vecteur d'éléments détaillés est ajouté à l'objet `Fault` en étant passé en paramètre à sa méthode `setDetailEntries()`.

Nous avons maintenant besoin de redéployer le service, en utilisant cette fois la classe `ModEcouteFauteSrcDonnees` comme valeur de l'élément `isd:FaultListener` dans le descripteur de déploiement :

```
<isd:faultListener>
javasoap.livre.ch7.ModEcouteFauteSrcDonnees
</isd:faultListener>
```

Si vous exécutez l'application cliente, la faute renvoyée par le serveur inclura désormais une section détails. Le client ne tient pas compte de cette section, mais nous pouvons constater le résultat de notre travail en examinant l'enveloppe SOAP renvoyée par le serveur :

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
 <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>Source de données indisponible</faultstring>
  <faultactor>/soap/servlet/rpcrouter</faultactor>
  <detail>
    <fournisseurSubstitution>www.mindstrm.com</fournisseurSubstitution>
    <descriptionCause>Panne électrique</descriptionCause>
    <contactSubstitution>rob@mindstrm.com</contactSubstitution>
    <codeCause>199</codeCause>
  </detail>
 </SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Les quatre champs détail sont inclus sous forme d'enfants de l'élément `detail`. Pour que cette information nous soit utile, il faudrait que l'application cliente accède à ces détails et nous allons nous y employer. Voici une version modifiée de l'application cliente qui extrait les détails envoyés avec la faute :

```
package javasoap.livre.ch7;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class ClientExemple {
```

```
    public static void main(String[] args)
       throws Exception {

       URL url =
         new URL(
           "http://localhost:8080/soap/servlet/rpcrouter");

       Call appel = new Call();

       appel.setTargetObjectURI("urn:ServiceFauteExemple");
       appel.setMethodName("genereFaute");
       try {
          Response rep = appel.invoke(url, "");
          if (rep.generatedFault()) {
             Fault faute = rep.getFault();
             String code = faute.getFaultCode();
             String desc = faute.getFaultString();
             System.out.println(code + ": " + desc);
             Vector v = faute.getDetailEntries();
             int cnt = v.size();
             for (int i = 0; i < cnt; i++) {
                Element n = (Element)v.elementAt(i);
                Node nd = n.getFirstChild();
                System.out.println(n.getNodeName() + ": " +
                                   nd.getNodeValue());
             }
          }
          else {
             Parameter ret = rep.getReturnValue();
             Object valeur = ret.getValue();
             System.out.println(valeur);
          }
       }
       catch (SOAPException e) {
          System.err.println("SOAPException (" +
                         e.getFaultCode() + "): " +
                         e.getMessage());
       }
    }
 }
```

Si `rep.generatedFault()` renvoie la valeur `true`, nous affichons le code et le libellé de la faute comme nous l'avons déjà fait. Nous appelons ensuite `faute.getDetailEntries()`, qui renvoie un vecteur contenant les éléments détaillés. Nous devons simplement itérer au sein du vecteur, transtyper chaque objet en une instance de `org.w3c.dom.Element` et invoquer la méthode `getFirstChild()` sur l'élément résultant. Nous affichons ensuite le nom de l'élément et les données du nœud. En exécutant l'exemple, vous devriez obtenir le résultat suivant :

```
SOAP-ENV:Server: Source de données indisponible
fournisseurSubstitution: www.mindstrm.com
```

```
descriptionCause: Panne électrique
contactSubstitution: rob@mindstrm.com
codeCause: 199
```

Envoyer et intercepter des exceptions sous GLUE

Les concepts concernant les fautes et les exceptions abordés sous Apache SOAP sont similaires sous GLUE, bien que certaines techniques diffèrent. En réalité, GLUE possède plusieurs techniques pour traiter les exceptions et si vous êtes intéressé par toutes les possibilités offertes, je vous encourage à lire le guide utilisateur car nous n'allons aborder ici qu'une seule technique.

Passons sur les notions élémentaires et prenons directement un exemple renvoyant des fautes avec une section de détails. GLUE contient sa propre classe d'exceptions SOAP appelée `electric.soap`. Elle diffère de la classe Apache `SOAPException` et vous ne devrez pas mélanger les deux. La classe `SOAPException` de GLUE vous permet de positionner le code de la faute, son libellé, son acteur et ses détails et possède plusieurs constructeurs qui utilisent une variété de combinaisons de ces éléments. De ce fait, vous n'aurez pas à à écrire de gestionnaire de faute pour récupérer les informations détaillées. Il suffira de renvoyer une instance de `SOAPException` possédant les valeurs adéquates. Créons un service nommé `urn:AutreServiceFaute` et implémentons-le par la classe `AutreServiceFaute`:

```
package javasoap.livre.ch7;
import electric.soap
import electric.xml.*;
public class AutreServiceFaute {
   public int genereFaute()
        throws SOAPException {
      Element elem = new Element("detail");
      Element sub = elem.addElement("codeCause");
      sub.addText("199");
      sub = elem.addElement("descriptionCause");
      sub.addText("Panne électrique");
      sub = elem.addElement("fournisseurSubstitution");
      sub.addText("www.mindstrm.com");
      sub = elem.addElement("contactSubstitution");
      sub.addText("rob@mindstrm.com");
      SOAPException e = new SOAPException(
                           "Source de données indisponible",
                           SOAPException.SERVER,
                           "urn:AutreServiceFaute",
                           elem);
      throw e;
   }
}
```

Dans cette classe, nous générons la section des détails avant de créer une instance de `SOAPException`. GLUE contient une API XML simple, que j'ai ici utilisé pour créer l'élé-

ment detail. La première étape est de créer une instance de electric.xml.Element avec le nom de balise "detail". Après cela, j'ajoute les nouveaux éléments à l'élément detail en appelant elem.addElement(), avec en paramètre le nom de balise de l'élément ajouté. Les données sont initialisées en appelant la méthode addText() de l'élément renvoyé par elem.addElement(). Une fois obtenu l'élément detail, nous sommes prêts à créer la SOAPException. Les paramètres du constructeur sont le libellé de la faute, son code, l'acteur et l'élément XML contenant les détails. Voici le code utilisé pour déployer urn:AutreServiceFaute:

```
package javasoap.livre.ch7;
import electric.util.Context;
import electric.server.http.HTTP;
import electric.registry.Registry;
public class AppServiceFaute {

   public static void main( String[] args )
                throws Exception {
      String ns = "urn:AutreServiceFaute";
      HTTP.startup("http://localhost:8004/glue");
      Context contexte = new Context();
      contexte.addProperty("activation", "application");
      contexte.addProperty("namespace", ns);
      Registry.publish(ns,
         javasoap.livre.ch7.AutreServiceFaute.class, contexte );
   }
}
```

L'interface Java de liaison à ce service contient la méthode genereFaute(), qui est déclarée pour renvoyer une instance de electric.soap:

```
package javasoap.livre.ch7;
import electric.soap;
public interface IAutreServiceFaute {
  int genereFaute() throws SOAPException;
}
```

GLUE prendra en charge les tâches d'envoyer la faute sur le réseau ainsi que de transformer cette faute en une instance de electric.soap.SOAPException côté client. Voici l'application cliente qui invoque la méthode genereFaute() de urn:AutreServiceFaute:

```
package javasoap.livre.ch7;
import electric.registry.RegistryException;
import electric.registry.Registry;
import electric.soap.SOAPException
import electric.xml.*;
public class ClientServiceFaute {
   public static void main(String[] args) throws Exception
   {
      try {
         IAutreServiceFaute srv =
           (IAutreServiceFaute)Registry.bind(
```

```
                "http://localhost:8004/glue/urn:AutreServiceFaute.wsdl",
                IAutreServiceFaute.class);
            srv.genereFaute();

        }
        catch (SOAPException se) {
            System.out.println(se.getSOAPCode());
            System.out.println(se.getMessage());
            System.out.println(se.getSOAPActor());
            System.out.println(se.getSOAPDetailElement());
        }
        catch (RegistryException e) {
            System.out.println(e);
        }
    }
}
```

Nous affichons le contenu de la faute dans le bloc `catch` de la `SOAPException`. Le code de faute, son libellé et l'acteur sont récupérés grâce aux appels sur la classe `SOAPException` des méthodes `getSOAPCode()`, `getMessage()` et `getSOAPActor()` et les détails peuvent être récupérés sous forme d'élément XML par un appel à `getSOAPDetailElement()`. Si nous voulions traiter les champs de la section détail, nous pourrions examiner les sous-éléments et leurs valeurs individuellement. Dans notre cas, nous nous contentons d'afficher le contenu en tirant parti de la méthode `toString()` de la classe `Element`. Si vous exécutez cet exemple, vous obtiendrez le résultat suivant :

```
Server
Source de données indisponible
urn:AutreServiceFaute
<detail>
  <codeCause>199</codeCause>
  <descriptionCause>Panne électrique</descriptionCause>
  <fournisseurSubstitution>www.mindstrm.com</fournisseurSubstitution>
  <contactSubstitution>rob@mindstrm.com</contactSubstitution>
</detail>
```

La seule chose inhabituelle est que le code de faute, affiché sur la première ligne, n'inclut pas son qualificateur d'espace de noms. Cette implémentation de `SOAPException` semble l'ignorer. C'est probablement mieux ainsi, dans la mesure où ce qualificateur n'est pas significatif sauf si vous connaissez sa valeur. Vous comprendrez ce que je veux dire en examinant l'enveloppe SOAP renvoyée par le serveur :

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <soap:Fault>
        <faultcode>soap:Server</faultcode>
        <faultstring>Source de données indisponible</faultstring>
```

```
        <faultactor>urn:AutreServiceFaute</faultactor>
        <detail>
          <codeCause>199</codeCause>
          <descriptionCause>Panne électrique</descriptionCause>
          <fournisseurSubstitution>www.mindstrm.com</fournisseurSubstitution>
          <contactSubstitution>rob@mindstrm.com</contactSubstitution>
        </detail>
      </soap:Fault>
    </soap:Body>
  </soap:Envelope>
```

La génération et le traitement des fautes font partie intégrante de l'écriture d'applications SOAP. Vous ne pouvez pas plus les ignorer que vous ne pouvez ignorer les exceptions et leur traitement en Java. Fort heureusement, les techniques pour les aborder sont simples et directes et avec un peu de travail supplémentaire, on peut y incorporer les détails souhaités.

On doit s'attendre à rencontrer des fautes dans le meilleur des logiciels. Le tout est de les prévoir à l'avance et de ne pas se laisser surprendre par des fautes imprévues. Ceci n'a rien à voir avec SOAP en particulier, mais avec la conception de logiciels en général. Concevez bien le traitement des fautes et vous n'aurez aucun mal à l'implémenter sous SOAP.

8

Autres techniques

Bien que SOAP soit principalement utilisé pour la communication de type RPC, il existe d'autres possibilités. Il peut également être utilisé pour créer des services de type message, dans lesquels une enveloppe SOAP est envoyée au serveur sans qu'une réponse ne soit requise. Le message peut contenir n'importe quelle charge utile appropriée, qui n'est pas obligatoirement un appel de procédure. Il s'agit là de quelque chose de similaire au courrier électronique. Lorsque vous envoyez un courrier, vous ne recevez rien en retour du serveur et il n'y a aucune règle quant au contenu du courrier. Le serveur accepte votre message et effectue son travail sans aucune autre intervention de l'expéditeur. Ceci est très différent des services RPC dans lesquels la valeur de retour est une partie intégrante du processus d'invocation de méthode.

Toutes les implémentations SOAP ne supportent pas les services de type message, mais ces services peuvent être des modèles utiles et méritent notre attention. GLUE ne les supporte pas mais Apache SOAP le fait. C'est donc ce dernier que nous utiliserons pour développer quelques exemples.

L'API Apache SOAP pour les services de type message est de plus bas niveau que celle que nous avons utilisée pour les RPC. Cela n'est pas très surprenant, dans la mesure où les RPC fonctionnent par transmission de messages. L'API RPC est donc bâtie au-dessus de l'API des messages et fournit un niveau d'abstraction qui rend relativement aisée l'écriture de services et de clients RPC. Cette facilité ne se retrouve pas dans le domaine des services de messages Apache SOAP. Nous devrons travailler avec des API de bas niveau qui nous feront manipuler directement l'assemblage et le désassemblage d'enveloppes, de structures XML et d'autres choses qui étaient automatiquement prises en charge dans le monde RPC. L'avantage est que nous contrôlerons mieux les processus, bien que dans la plupart des cas, je préfère de bonnes abstractions.

Dans la plupart des exemples vus jusqu'à présent, nous avons utilisé le style d'encodage SOAP, mais ceci n'est pas la seule façon d'encoder les données. L'encodage littéral est un autre mécanisme utile et important, particulièrement lorsque vous voulez transmettre des documents XML existant à l'aide de SOAP. Voyons donc un exemple d'utilisation de cet encodage afin de voir comment utiliser cette option.

Nous allons également examiner les attachements sous SOAP. L'idée sous-jacente est simple : tout comme on peut attacher un fichier à un courrier électronique, on peut attacher des données externes à un message SOAP à l'aide de MIME. Cette technique est utile lorsque vous souhaitez transmettre des données qui n'ont pas à être encodées ou désencodées dans votre message SOAP, comme par exemple une image ou un fichier son. L'utilisation d'attachements permet d'envoyer de grands objets binaires avec des messages SOAP sans avoir à inclure les données dans l'enveloppe SOAP. Dans la plupart des systèmes, l'utilisation des attachements est bien plus performante que l'encodage de données dans l'enveloppe.

Système de messages SOAP

Le format d'une enveloppe SOAP destinée à des services de messages diffère de celui utilisé pour les services RPC. On n'y trouvera pas de concept de signature ou de paramètres de méthode ni de valeurs de retour. Globalement, on envoie du XML à un service. Il existe toutefois quelques prérequis qui permettent d'envoyer le message au bon service et à la bonne méthode. Le nom du premier élément enfant du `Body` SOAP doit correspondre au nom de la méthode de service utilisée et l'attribut d'espace de noms `xmlns` de cet élément spécifie le nom du service. Voici un message SOAP simple, envoyé à la méthode `enregistreTemperature` du service `urn:RegistreMeteo` :

```
<SOAP_ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
      <enregistreTemperature xmlns="urn:RegistreMeteo"/>
      <temperature>75.5</temperature>
      <codepostal>50328</codepostal>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

La méthode de service `enregistreTemperature` enregistre la température courante associée à un code postal. Vous noterez qu'il n'y a pas de données dans l'élément `enregistreTemperature`, bien qu'il serait parfaitement valide qu'il y en ait, de même que des éléments enfants. Mais cela n'est tout simplement pas requis. En réalité, j'aurais pu faire des éléments `temperature` et `codepostal`, des enfants d'`enregistreTemperature` plutôt que des enfants d'un même parent. Les deux approches sont valides. Pour SOAP, tout ceci n'est qu'un message envoyé à une méthode et non un appel de procédure contenant des arguments à traiter. Comme vous le verrez bientôt, nous allons devoir écrire du code pour accéder d'une façon ou d'une autre à ces éléments.

Tant que ce premier élément est correctement positionné, le reste du XML est laissé à votre entière discrétion. Vous n'avez aucun paramètre à encoder ni de contraintes de syntaxe à respecter. Cela ne veut pas dire pour autant que vous pouvez envoyer n'importe quel message à n'importe quel service. L'absence de contraintes strictes de syntaxe rend au contraire plus difficile l'interopérabilité entre des services et des clients qui ne sont pas sous votre contrôle. Vous devez vous assurez que vos messages se conforment aux formats requis par de nombreux autres services. Cela n'enlève rien à ce modèle de service et le rend même plus flexible, avec tous les inconvénients associés à cette flexibilité.

Écrire un service de messages

L'écriture d'un service de type message n'est pas très différente de celle d'un service RPC : vous devrez écrire une méthode appelée lorsque le message est reçu. La différence réside dans la signature de la méthode. Avec les RPC, nous avions le luxe de pouvoir définir les paramètres de la méthode de service ainsi que leur type et les valeurs de retour. Dans le modèle de messages, toutes les méthodes de service possèdent la même signature. Il existe trois paramètres.

Le premier est une instance de `org.apache.soap.Envelope`, qui encapsule le contenu de l'enveloppe SOAP reçue par la méthode de service. Les deux autres paramètres sont des instances de `org.apache.soap.rpc.SOAPContext`,[1] une classe qui encapsule un message MIME multiparties. L'un est pour l'entrée, l'autre pour la sortie. J'ai bien dit la sortie ! Bien que les services de messages SOAP ne soient pas obligés de fournir une réponse, il peuvent le faire si la couche de transport sous-jacente le permet. HTTP le permettant, un service de messages bâti sur ce protocole pourra fournie une réponse. En réalité, Apache SOAP oblige un service de messages à fournir une réponse.

Écrivons une classe Java pour le service `urn:RegistreMeteo`. Nous allons avoir besoin d'une méthode nommée `enregistreTemperature` de façon à ce que le client puisse envoyer la température courante au service. Voici la classe `javasoap.livre.ch8.RegistreMeteo` qui implémente le service :

```
package javasoap.livre.ch8;
import java.io.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.SOAPContext;
import javax.mail.MessagingException;
import org.w3c.dom.*;
public class RegistreMeteo {
   Hashtable _register = new Hashtable();
   public void enregistreTemperature (Envelope env, SOAPContext
                     reqCtx, SOAPContext repCtx)
            throws Exception {

      Vector v = env.getBody().getBodyEntries();
      int cnt = v.size();
      String codepostal = null;
      String temperature = null;
      for (int i = 0; i < cnt; i++) {
         Element e = (Element)v.elementAt(i);
         String nom = e.getTagName();
         if (nom.equals("codepostal")) {
            codepostal = e.getFirstChild().getNodeValue();
         }
         else if (nom.equals("température")) {
```

1. Il est intéressant de voir que cette classe se trouve dans le package SOAP RPC. Cela ne signifie pas qu'elle est destinée à être utilisée uniquement dans les services de type RPC. Elle devrait probablement se trouver ailleurs.

```
                temperature = e.getFirstChild().getNodeValue();
            }
        }
        if (codepostal == null || temperature == null) {
            throw new IllegalArgumentException(
                    "Code Postal et/ou Température non spécifiés");
        }
        _register.put(codepostal, temperature);
        repCtx.setRootPart("OK", "text/xml");
    }
}
```

Nous devons accéder aux éléments contenus dans le corps SOAP. Nous appelons pour cela la méthode `getBody()` du paramètre `env`, qui renvoie une instance de `org.apache.soap.Body`. Nous appelons ensuite la méthode `getBodyEntries()` de l'objet corps, qui renvoie un `java.util.Vector` contenant des instances de `org.w3c.dom.Element`. Chaque `Element` du vecteur encapsule l'un des éléments descendant direct du corps SOAP. Si vous examinez l'enveloppe SOAP que nous envoyons au service, vous verrez que nous avons utilisé trois descendants directs du corps : `enregistreTemperature`, `codepostal` et `temperature`. C'est le format que nous allons utiliser pour la méthode `enregistreTemperature` du service `urn:RegistreMeteo`. Nous itérons au sein du `Vecteur` pour récupérer chaque `Element` du corps. Pour déterminer quel type d'élément nous avons récupéré, nous appelons la méthode `getTagName()` et vérifions le résultat pour nous assurer qu'il s'agit de l'élément `codepostal` ou `temperature`. (Nous allons également récupérer l'élément `enregistreTemperature` utilisé pour spécifier le service et la méthode, mais nous n'en aurons pas l'usage dans cet exemple.) Dès que nous récupérons un élément qui nous intéresse, nous positionnons son premier (et seul) nœud enfant, dans lequel nous trouverons les données de cet élément. Les données sont stockées dans les variables locales `codepostal` et `temperature`. Une fois sortis de la boucle, nous vérifions si l'une de ces variables a gardé la valeur `null`. Si c'est le cas, au moins l'un des éléments requis n'était pas présent dans le message et nous renvoyons une exception. Celle-ci conduit Apache SOAP en renvoyer une faute SOAP à l'appelant. Dans le cas contraire, les données sont stockées dans la table de hachage `_registre`, le code postal servant de clé et la température de valeur. Nous allons déployer ce service en utilisant une portée de type application, ainsi le contenu de la table de hachage restera constant au gré des appels.

La dernière étape consiste à positionner une valeur de retour. Cela peut sembler étonnant. Pourquoi un service de type message devrait-il renvoyer une valeur ? Parce qu'Apache envoie une exception si cela n'est pas fait. Comme je l'ai déjà indiqué, Apache oblige à fournir une réponse avec un contenu. La façon la plus simple pour positionner une valeur de retour est de fournir un simple texte comme racine du contexte de réponse en appelant pour cela la méthode `repCtx.setRootPart()`. Le premier paramètre est le texte à envoyer. La seconde partie est le type MIME des données, dans notre cas `text/xml`. Je ne fais que renvoyer la chaîne « OK » pour indiquer que les données ont été reçues.

Déployons maintenant le service. Voici le descripteur de déploiement :

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
```

```
      id="urn:RegistreMeteo" type="message">
    <isd:provider
       type="java"
       scope="Application"
       methods="enregistreTemperature">
      <isd:java
         class="javasoap.livre.ch8.RegistreMeteo"
         static="false"/>
    </isd:provider>

    <isd:faultListener>org.apache.soap.server.DOMFaultListener
    </isd:faultListener>
    <isd:mappings>
    </isd:mappings>
  </isd:service>
```

La seule différence entre ce descripteur et ceux créés pour les services de type RPC se trouve dans l'utilisation de l'attribut type dans l'élément `isd:service`. La valeur de cet attribut est maintenant "message", indiquant en cela que le service déployé est de ce type et non de type RPC.

Ce service est donc prêt à recevoir d'applications clientes des mises à jour de température depuis des endroits divers. Examinons l'application cliente qui envoie ces mises à jour. Nous allons construire le XML de l'enveloppe SOAP puis l'envoyer au service `urn:RegistreMeteo`.

```
package javasoap.livre.ch8;
import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.apache.soap.*;
import org.apache.soap.messaging.*;
import org.apache.soap.transport.*;
import org.apache.soap.util.xml.*;
public class ClientMeteo {
   public static void main (String[] args) throws Exception {
      DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder();
      Document doc = xdb.newDocument();

      Element elem =
            doc.createElementNS(Constants.NS_URI_SOAP_ENV,
            "SOAP-ENV:Envelope");
      doc.appendChild(elem);

      Element sous_elem =
            doc.createElementNS(Constants.NS_URI_SOAP_ENV,
            "SOAP-ENV:Body");
      elem.appendChild(sous_elem);
      elem = sous_elem;
```

```
        sous_elem = doc.createElement("enregistreTemperature");
        sous_elem.setAttribute("xmlns", "urn:RegistreMeteo");
        elem.appendChild(sous_elem);
        sous_elem = doc.createElement("codepostal");
        Text txt = doc.createTextNode("");
        txt.setData("12345");
        sous_elem.appendChild(txt);
        elem.appendChild(sous_elem);
        sous_elem = doc.createElement("temperature");
        txt = doc.createTextNode("");
        txt.setData("52.3");
        sous_elem.appendChild(txt);
        elem.appendChild(sous_elem);
        Envelope msgEnv = Envelope.unmarshall(
                          doc.getDocumentElement());
        URL url = new URL(
          "http://localhost:8080/soap/servlet/messagerouter");
        Message msg = new Message();
        msg.send(url, "", msgEnv);

        SOAPTransport st = msg.getSOAPTransport();
        BufferedReader br = st.receive();
        String ligne;
        while ((ligne = br.readLine()) != null) {
           System.out.println(ligne);
        }
    }
}
```

L'essentiel de ce code ne concerne pas vraiment SOAP mais la construction d'un document XML qui se trouve être une enveloppe SOAP. Il me semble toute fois que si vous envisagez d'utiliser les services de type messages Apache, vous aurez besoin de créer vos enveloppes à la volée.[2] Voyons donc rapidement comment procéder.

La première étape consiste à obtenir une instance de `org.w3c.dom.Document`, qui encapsule entièrement le document XML en commençant par l'élément racine. Nous obtenons le `Document` initial en appelant `XMLParserUtils.getXMLDocBuilder()` pour récupérer un `DocumentBuilder`, puis appelons sa méthode `newDocument()`. Nous pouvons maintenant travailler sur l'élément enveloppe, que nous allons utiliser comme racine de notre document. La méthode `doc.createElementNS()` crée un élément ainsi que son identifiant d'espace de noms. Le premier paramètre est l'espace de noms standard pour les enveloppes SOAP `http://schemas.xmlsoap.org/soap/envelope/`, que nous avons obtenu à partir de `Constants.NS_URI_SOAP_ENV`. Le second paramètre est un nom qualifié, c'est-à-dire un qualificateur et un nom d'élément séparés par deux points (:). Dans ce cas, nous utilisons le qualificateur `SOAP-ENV` pour rester cohérents avec nos exemples précédents, mais le qualificateur n'est rien de plus qu'un identifiant d'espace

2. Si vous ne créez pas votre propre enveloppe à la volée, vous pouvez jeter un œil à l'exemple de message fourni avec Apache SOAP. cet exemple suppose que l'enveloppe a été générée par un autre système et vous montre comment la charger dans votre application pour l'envoyer à un service.

de noms et vous pouvez utiliser n'importe quel nom. Le nom de l'élément lui-même est `Envelope`. L'enveloppe SOAP créée, nous devons l'ajouter au document en appelant `doc.appendChild()`. Nous utilisons la même technique pour créer l'élément `Body`, mais cette fois nous l'ajoutons à l'élément `Envelope` plutôt qu'au document lui-même, car le `Body` SOAP est un élément enfant de `Envelope`.

Les éléments `Envelope` et `Body` étant positionnés, nous devons créer les éléments pour notre contenu spécifique et les ajouter au corps. Nous lions pour cela la variable `elem` à l'élément `Body` et utilisons la variable `sous_elem` pour créer les enfants du corps. Pour invoquer la bonne méthode de service, le premier élément enfant du corps doit être nommé `enregistreTemperature`. Cet élément est créé en appelant `doc.createElement()` et en passant en paramètre le nom de l'élément. L'élément résultant est sauvegardé dans la variable `sous_elem`. Pour invoquer le bon service, nous devons donner comme valeur à l'attribut `xmlns` le nom du service. Nous faisons cela en appelant `sous_elem.setAttribute()`, avec comme premier paramètre le nom de l'attribut et comme second sa valeur. L'élément `enregistreTemperature` ne possédant ni données ni éléments enfants, nous n'avons donc qu'à l'ajouter au corps en appelant `elem.appendChild(sous_elem)`.

Nous créons ensuite l'élément `codepostal`, en utilisant de nouveau la méthode `doc.createElement()`. Aucun attribut n'est nécessaire pour cet élément, mais il nous faut ajouter un nœud pour le code postal. Nous créons une instance de `org.w3c.dom.Text` en appelant la méthode `doc.createTextNode()`. La classe `Text` est utilisée pour représenter des données textuelles, ce qui est parfait pour notre usage. Le code postal est positionné en appelant `txt.setData()` et le nœud est ajouté à l'élément `codepostal` par la méthode `sous_elem.appendChild()`. L'élément `codepostal` est à son tour ajouté à l'élément `Body`. Nous créons l'élément `temperature` exactement de la même façon et l'ajoutons également à l'élément `Body`.

Nous allons envoyer ce message à l'aide d'une instance de `org.apache.soap.messaging.Message`, mais nous devons d'abord transformer le document XML en une instance de `org.apache.soap.Envelope`. Pour cela, nous appelons la méthode `Envelope.unmarshall()` en lui passant le document racine. Cela semble ajouter une charge supplémentaire. Vous pouvez être sûr cependant que l'objet `Envelope` qui résultera sera assemblé pour être envoyé au serveur. Dans tous les cas, le résultat de l'appel à cette méthode sera une instance d'`Envelope` nommée `msgEnv`. Nous créons ensuite une instance de `java.net.URL` avec l'adresse du routeur SOAP prenant en charge l'acheminement des services de messages. Pour les services RPC, nous envoyions les requêtes SOAP à *http://localhost:8080/soap/servlet/rpcrouter*. Pour les services de messages, nous les enverrons à *http://localhost:8080/soap/servlet/messagerouter*. Nous pouvons maintenant créer une instance de `Message` et appeler sa méthode `send()`. L'URL est le premier paramètre et l'enveloppe le dernier. Le second est une chaîne représentant l'acteur SOAP. Dans la mesure où nous n'en avons pas l'usage ici, nous laisserons la chaîne vide.

Puisque nous savons que la couche de transport de notre service est HTTP et qu'il répond soit par une faute, soit par un simple « OK », il est utile de jeter un œil à la réponse avant que l'application ne se termine. Pour voir la réponse, nous commençons par appeler `msg.getSOAPTransport()`, qui nous donne une instance de la couche de transport utilisée pour envoyer le message. Cette méthode renvoie une instance de `org.apache.soap.transport.SOAPTransport`. Si la couche de transport supporte les réponses,

comme le fait HTTP, vous pouvez appeler sa méthode `receive()`, qui renvoie un objet `BufferedReader`. Si la couche ne supporte pas les réponses, la méthode `receive()` renvoie `null`. Une fois obtenu le `BufferedReader`, tout ce qui nous reste à faire est de récupérer chaque ligne de la réponse et de l'afficher sur la console. Si le service s'est satisfait du message, la seule chose à afficher sera la chaîne « OK ». Autrement, c'est la faute SOAP qui apparaîtra.

Examinons maintenant l'enveloppe SOAP envoyée par le client :

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <enregistreTemperature xmlns="urn:RegistreMeteo"/>
    <codepostal>12345</codepostal>
    <température>52.3</température>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Nous avons maintenant un service capable de collecter de l'information et un client capable d'en envoyer, mais rien ne nous permet pour l'instant de récupérer l'information collectée. Il nous faut étendre le service par l'ajout d'une méthode permettant d'obtenir la température pour un code postal donné. Attendez... Cela ressemble plus à un service de type RPC qu'à un service de messages. C'est une chose d'avoir un service de messages qui répond quelque chose comme « OK, j'ai bien reçu votre message », mais s'en est une autre de le faire répondre par un résultat basé sur le contenu du message reçu. Pourquoi devriez-vous utiliser les services de type RPC lorsque vous travaillez à l'aide du modèle requête/réponse ? Beaucoup de développeurs ont une opinion bien arrêtée sur le type de service approprié à chaque application, mais hormis le fait que toutes les couches de transport ne supportent pas l'envoi de réponses, il n'y a aucune raison de ne pas utiliser un service de messages dans cette situation. Il est facile de tomber dans le piège qui consiste à ne réfléchir que selon le modèle procédural. Certains systèmes peuvent bénéficier de la liberté apportée par le modèle messages, dans lequel vous n'êtes pas limité par des règles de passage de paramètres spécifiques comme vous l'êtes dans le monde RPC. Vous pourriez vouloir par exemple envoyer une demande d'achat à un service de messages et recevoir en retour un document de confirmation indiquant la date de livraison. Cet échange pourrait être implémenté par RPC, mais il est bien plus facile de l'aborder sous l'angle d'un échange de messages. Bien que la structure du RPC SOAP ne vous empêche pas de développer des modèles sans envoi de paramètres, le modèle messages sera bien plus approprié dans ce cas. Étendons donc notre service par une méthode `getTemperature()` renvoyant un document XML.

```
package javasoap.livre.ch8;
import java.io.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.SOAPContext;
import javax.mail.MessagingException;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
public class RegistreMeteo {
   Hashtable _registre = new Hashtable();
```

```
Hashtable _dateheure = new Hashtable();
public void getTemperature(Envelope env, SOAPContext reqCtx,
                                    SOAPContext repCtx)
     throws Exception {
   Vector v = env.getBody().getBodyEntries();
   int cnt = v.size();
   String codepostal = null;
   for (int i = 0; i < cnt; i++) {
      Element e = (Element)v.elementAt(i);
      String name = e.getTagName();
      if (name.equals("codepostal")) {
         codepostal = e.getFirstChild().getNodeValue();
      }
   }
   if (codepostal == null) {
      throw new IllegalArgumentException(
                 "Code Postal non spécifié");
   }
   String temperature = (String)_registre.get(codepostal);
   String heureenreg = (String)_dateheure.get(codepostal);
   String reponse = "<RegistreMeteoResponse>" +
                    (char)10 +
                    "   <temperature>" + temperature +
                    "</temperature>" +
                    (char)10 +
                    "   <heureEnreg>" + heureenreg +
                    "</heureEnreg>" +
                    (char)10 +
                    "</RegistreMeteoResponse>";
   repCtx.setRootPart(reponse, "text/xml");
}
public void enregistreTemperature (Envelope env, SOAPContext
                 reqCtx, SOAPContext repCtx)
     throws Exception {

   Vector v = env.getBody().getBodyEntries();
   int cnt = v.size();
   String codepostal = null;
   String temperature = null;
   for (int i = 0; i < cnt; i++) {
      Element e = (Element)v.elementAt(i);
      String name = e.getTagName();
      if (name.equals("codepostal")) {
         codepostal = e.getFirstChild().getNodeValue();
      }
      else if (name.equals("temperature")) {
         temperature = e.getFirstChild().getNodeValue();
      }
   }
   if (codepostal == null || temperature == null) {
```

```
            throw new IllegalArgumentException(
                  "Code Postal et/ou Temperature non spécifiés");
         }
         _registre.put(codepostal, temperature);
         _dateheure.put(codepostal, new Date().toString());
         repCtx.setRootPart("OK", "text/xml");
      }
   }
```

Pour rendre la réponse plus intéressante, nous avons modifié la méthode enregistreTemperature() pour stocker dans une autre table de hachage nommée _dateheure, la date et l'heure à laquelle la température a été enregistrée, en utilisant le code postal comme clé et la date et l'heure courantes comme valeur. Mai l'ajout le plus important est la méthode de service getTemperature(). Cette méthode commence par rechercher dans les éléments enfants du corps SOAP, un élément nommé codepostal, qui indique à la méthode quel code postal est demandé. Si aucun élément codepostal n'est trouvé, la méthode renvoie une exception qui produit une faute SOAP. Une fois le code postal obtenu, on récupère la température et la date et l'heure d'enregistrement à partir des tables de hachage _registre et _dateheure. On construit ensuite ensuite un document XML dans la variable String nommée reponse. Ce document est fort simple. Sa racine est un élément nommé RegistreMeteoResponse, contenant deux éléments enfants nommés température et heureEnreg. Ces éléments contiennent la température et la date et l'heure d'enregistrement pour le code postal spécifié. La chaîne reponse est utilisée comme premier paramètre de la méthode repCtx.setRootPart(). Le résultat est que nous avons répondu au message entrant par un document XML contenant les données demandées. Vous allez devoir redéployer le service en ajoutant la méthode getTemperature() au descripteur de déploiement.

Créons maintenant une application cliente demandant des informations sur la température. Ce client procédera de la façon que nous avons vu dans les autres exemples. Cette fois, nous n'incluons que l'élément codepostal et ses données et le premier élément est nommé getTemperature pour invoquer la bonne méthode de service. Voici le code du client:

```
package javasoap.livre.ch8;
import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.apache.soap.*;
import org.apache.soap.messaging.*;
import org.apache.soap.transport.*;
import org.apache.soap.util.xml.*;
public class ClientTemperature {
   public static void main (String[] args) throws Exception {
      DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder();
      Document doc = xdb.newDocument();

      Element elem =
               doc.createElementNS(Constants.NS_URI_SOAP_ENV,
```

```
                "SOAP-ENV:Envelope");
        doc.appendChild(elem);

        Element sous_elem =
                doc.createElementNS(Constants.NS_URI_SOAP_ENV,
                "SOAP-ENV:Body");
        elem.appendChild(sous_elem);
        elem = sous_elem;
        sous_elem = doc.createElement("getTemperature");
        sous_elem.setAttribute("xmlns", "urn:RegistreMeteo");
        elem.appendChild(sous_elem);
        sous_elem = doc.createElement("codepostal");
        Text txt = doc.createTextNode("");
        txt.setData("12345");
        sous_elem.appendChild(txt);
        elem.appendChild(sous_elem);
        Envelope msgEnv = Envelope.unmarshall
                          (doc.getDocumentElement ());
        URL url = new URL(
            "http://localhost:8080/soap/servlet/messagerouter");
        Message msg = new Message ();
        msg.send (url, "", msgEnv);

        SOAPTransport st = msg.getSOAPTransport ();
        BufferedReader br = st.receive ();
        String ligne;
        while ((ligne = br.readLine ()) != null) {
           System.out.println (ligne);
        }
     }
  }
```

Avant d'exécuter cette application, vous devrez alimenter le service par des données de température pour le code postal 12345. (Redéployer le service avec la nouvelle méthode `getTemperature()` détruit les données sauvegardées.) Après avoir fourni quelques données au service, exécutez le client. Il devrait renvoyer ceci :

```
<RegistreMeteoResponse>
   <température>52.3</température>
   <heureEnreg>Wed Oct 31 14:58:48 EST 2001</heureEnreg>
</RegistreMeteoResponse>
```

Nous n'allons pas nous occuper de l'enveloppe SOAP envoyée à la méthode `getTemperature()` dans la mesure où vous pouvez deviner à quoi elle ressemble. La seule partie intéressante est la réponse.

Il serait appréciable de structurer un peu le résultat. Pour l'instant, nous n'avons fait que renvoyer quelques éléments XML. Élaborons donc un document avec cette réponse. Le code suivant remplace tout se qui se trouve après l'appel à `msg.send()`. Une fois obtenu le `BufferedReader`, nous avons toujours besoin d'une instance de `DocumentBuilder`. Mais à présent, au lieu de fabriquer un document morceau par morceau, nous allons utiliser le contenu de la réponse. Nous créons une instance de `org.xml.sax.InputSour-`

ce, en envoyant BufferedReader au constructeur. Nous appelons ensuite xdb.parse() avec l'InputSource en paramètre. Cela provoque l'analyse du XML et fabrique un instance de Document. Après avoir récupéré l'élément racine, nous l'affichons afin d'être certains d'avoir récupéré l'élément que nous attendions. Nous obtenons alors une liste de tous les éléments enfants de la racine en appelant getElementsByTagName("*") et en lui passant le caractère joker (*). Cette méthode récupère tous les éléments enfants de l'élément racine RegistreMeteoResponse. Finalement, la liste est parcourue et chaque élément est affiché ainsi que ses données associées.

```
SOAPTransport st = msg.getSOAPTransport ();
BufferedReader br = st.receive ();

xdb = XMLParserUtils.getXMLDocBuilder();
InputSource is = new InputSource(br);
doc = xdb.parse(is);
elem = doc.getDocumentElement();
System.out.println("L'élément racine est : " +
                            elem.getTagName());

NodeList enfant = elem.getElementsByTagName("*");
int cnt = enfant.getLength();
for (int i = 0; i < cnt; i++) {
   Node n = enfant.item(i);
   String nom = n.getNodeName();
   String val = n.getFirstChild().getNodeValue();
   System.out.println(nom + " : " + val);
}
```

L'exécution du client modifié produit le résultat suivant:

```
L'élément racine est : RegistreMeteoResponse
température: 52.3
heureEnreg: Wed Oct 31 14:58:48 EST 2001
```

On voit donc qu'en manipulant un peu de XML, on peut renvoyer des données significatives depuis un service de type messages, sous réserve que la couche de transport sous-jacente permette des réponses.

Encodage littéral

Nous avons vu comment utiliser SOAP pour les services de type RPC ainsi que la façon de transmettre du XML dans le monde moins restrictif des services de type RPC. Une autre possibilité est de transmettre du XML à l'aide de services RPC. Le problème est que les services RPC se basent sur un mécanisme de mise en correspondance de types pour effectuer la conversion entre des éléments XML et des classes Java. Il serait plaisant de pouvoir utiliser une invocation de méthode de type RPC pour envoyer des paramètres et d'obtenir en résultat du XML standard. Et bien cela est tout à fait possible. Pour cela, vous devrez utiliser l'encodage littéral XML, qui indique à SOAP que les données sont du XML et doivent le rester.

Revenons au monde de la finance. Nous allons créer un service qui fournira des informations au format XML sur des entreprises. Le service s'appelle urn:ServiceDonnees-

`Entreprise` et possède une méthode nommée `getDonneesSymbole`. Le service est de type RPC et la méthode reçoit en paramètre un symbole d'action. Les données renvoyées peuvent comprendre le nom de la société, son adresse, l'année de création, le nombre d'employés et les valeurs maximum et minimum de l'action au cours des 52 semaines précédentes. Ce service tirera ses données d'un autre système qui contient déjà dans une base de données des informations sur les entreprises au format XML. Il ne sera donc pas du ressort de notre service de mettre en forme les données mais seulement de les renvoyer à l'appelant. Ce scénario est idéal pour l'utilisation de l'encodage littéral XML. Si le service devait traiter les données d'une façon quelconque, il devrait juste être conçu pour renvoyer un type complexe qui puisse être mis en correspondance avec une classe Java. Une autre raison d'utiliser l'encodage littéral est que l'application cliente peut vouloir récupérer les données au format XML utilisé par le serveur de données. Ce serait une perte de temps de convertir le XML côté serveur pour voir le client le convertir à nouveau. Dans ce cas, nous utilisons SOAP pour fournir un accès à des données existantes et se baser sur un système existant pour effectuer les traitements.

Pour créer une méthode Java comportant une signature appropriée, nous avons besoin d'une classe encapsulant un document XML, dans la mesure où c'est ce que nous voulons renvoyer à l'appelant. `org.w3c.dom.Element` est parfait pour cette tâche et nous allons donc définir notre méthode de service de façon à renvoyer une instance de cette classe. La classe `javasoap.livre.ch8.ServiceDonneesEntreprise` qui suit implémente le service. Celui-ci ouvre un fichier dont le nom est le symbole passé en paramètre suivi de l'extension *.xml*. Le fichier est supposé se trouver dans le même répertoire que la classe elle-même. (Nous supposons qu'un système quelconque a déjà généré les fichiers XML de données d'entreprise que notre service va charger.) Un `java.io.FileReader` est créé en utilisant le nom de fichier comme paramètre pour son constructeur. Nous créons ensuite une instance de `DocumentBuilder` et appelons sa méthode `parse()` pour transformer le contenu du fichier en un document XML. Le paramètre envoyé à la méthode `parse()` est une instance de `org.xml.sax.InputSource`, que nous créons en utilisant le `FileReader` comme paramètre du constructeur. Tout ce qu'il nous reste à faire est de renvoyer l'élément racine que nous pouvons récupérer en appelant `doc.getDocumentElement()`. Vous pouvez déployer ce service à l'instar des autres services de ce livre, `getDonneesSymbole` étant l'unique méthode de service. Aucune mise en correspondance particulière n'est nécessaire.

```
package javasoap.livre.ch8;
import java.io.*;
import java.util.*;
import org.apache.soap.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
public class ServiceDonneesEntreprise {
   public ServiceDonneesEntreprise() {
   }
   public Element getDonneesSymbole(String symbole)
         throws Exception {
      FileReader fr = new FileReader(symbole + ".xml");
      DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder();
```

```
        Document doc = xdb.parse(new InputSource (fr));
        if (doc == null) {
           throw new SOAPException(Constants.FAULT_CODE_SERVER,
                                   "Données invalides");
        }
        return doc.getDocumentElement();
     }
  }
```

Voyons un exemple de fichier de données d'entreprise contenant un échantillon de XML que notre service renverra à l'appelant :

```
<donneesEntreprise>
   <symbole>MINDSTRM</symbole>
   <nom>MindStream Software, Inc.</nom>
   <adresse>
      <adresse1>111 Smithtown Bypass</adresse1>
      <adresse2>Suite 208</adresse2>
      <ville>Hauppauge</ville>
      <etat>NY</etat>
      <codepostal>11788</codepostal>
   </adresse>
   <anneeCreation>1998</anneeCreation>
   <nbEmployes>inconnu</nbEmployes>
   <historiqueCotation>
      <maxAnnuel>75.5</maxAnnuel>
      <minAnnuel>60</minAnnuel>
   </historiqueCotation>
</donneesEntreprise>
```

Nous pouvons maintenant écrire un client pour invoquer la méthode getDonneesSymbole :

```
package javasoap.livre.ch8;
import java.io.*;
import java.util.*;
import java.net.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
public class ClientDonnees {
   public static void main(String[] args)
         throws Exception {
      URL url = new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");
      Call appel = new Call();
      appel.setTargetObjectURI("urn:ServiceDonneesEntreprise");
      appel.setMethodName("getDonneesSymbole");
      appel.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
```

```
        String symbole = "MINDSTRM";
        Vector params = new Vector();
        params.addElement(new Parameter("symbole", String.class,
                    symbole, Constants.NS_URI_SOAP_ENC));
        appel.setParams(params);
        Response rep;
        try {
           rep = appel.invoke(url, "");
        }
        catch (SOAPException e) {
           System.out.println(e.getMessage());
           return;
        }
        if (!rep.generatedFault()) {
           Parameter ret = rep.getReturnValue();
           Element bookEl = (Element)ret.getValue();
           System.out.println(DOM2Writer.nodeToString(bookEl));
        }
     }
  }
```

Nous devons utiliser l'encodage XML littéral comme style d'encodage car nous voulons d'une part que le serveur sérialise sa réponse sous forme de document XML littéral et d'autre part parce que nous voulons que la réponse soit interprétée de cette façon. Pour cela, nous appelons `appel.setEncodingStyleURI()` avec comme argument `Constants.NS_URI_LITERAL_XML`. Sous SOAP Apache, ceci positionne le style d'encodage de l'élément `Body`. C'est le niveau le plus haut auquel l'attribut `encodingStyle` peut être positionné. Toutefois, il ne faut pas que le paramètre que nous envoyons (le symbole de l'action) soit interprété comme du XML. Ce n'est après tout qu'une simple chaîne de caractères. Nous voulons utiliser l'encodage SOAP standard pour le paramètre et donc que l'attribut `encodingStyle` apparaisse également dans l'élément `symbole` avec une valeur d'encodage SOAP appropriée. Pour spécifier le style d'encodage de ce paramètres, nous positionnons le quatrième argument du constructeur `Parameter` à `Constants.NS_URI_SOAP_ENC`. Dans tous les exemples précédents, nous avons positionné cet attribut à `null`, ce qui signifiait que l'attribut `encodingStyle` ne devait pas apparaître au niveau des paramètres. Le style d'encodage pour les paramètres est par défaut celui du corps. Le positionner pour l'élément `symbole` supplante le style d'encodage du corps.

Nous pouvons maintenant invoquer la méthode comme à l'accoutumée mais nous devons transtyper la valeur de retour en une instance d'`Element`, dans la mesure où nous nous attendons à ce qu'elle soit du XML littéral. En passant l'`Element` résultant à la méthode `DOM2Writer.nodeToString()`, nous pouvons afficher le XML sur la console. Examinons l'enveloppe SOAP envoyée au service :

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getDonneesSymbole
xmlns:ns1="urn:ServiceDonneesEntreprise"
```

```
SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
<symbole xsi:type="xsd:string"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
MINDSTRM
</symbole>
</ns1:getDonneesSymbole>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Comme vous pouvez le voir, l'attribut encodingStyle est positionné de façon à effectuer un encodage XML littéral pour l'élément SOAP Body. L'élément symbole supplante cela et utilise l'encodage SOAP. Voici maintenant l'enveloppe de la réponse. Vous pouvez voir que la valeur de retour utilise l'encodage littéral XML, la réponse étant entièrement du XML.

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getDonneesSymboleResponse
xmlns:ns1="urn:ServiceDonneesEntreprise"
SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
<return>
<donneesEntreprise>
   <symbole>MINDSTRM</symbole>
   <nom>MindStream Software, Inc.</nom>
   <adresse>
      <adresse1>111 Smithtown Bypass</adresse1>
      <adresse2>Suite 208</adresse2>
      <ville>Hauppauge</ville>
      <etat>NY</etat>
      <codepostal>11788</codepostal>
   </adresse>
   <anneeCreation>1998</anneeCreation>
   <nbEmployes>unknown</nbEmployes>
   <historiqueCotation>
      <maxAnnuel>75.5</maxAnnuel>
      <minAnnuel>60</minAnnuel>
   </historiqueCotation>
</donneesEntreprise>
</return>
</ns1:getDonneesSymboleResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Vous pouvez utiliser la même technique si vous souhaitez envoyer du XML littéral en paramètre à une méthode de service RPC. Spécifiez simplement le style d'encodage `Constants.NS_URI_LITERAL_XML` lorsque vous appelez le constructeur `Parameter`. Il n'est pas obligatoire d'utiliser le même style d'encodage pour tous les paramètres. Faites ce qui est le plus approprié à votre application.

SOAP et les attachements

De nombreuses implémentations SOAP permettent d'envoyer sous la forme d'un attachement, des données accompagnant l'enveloppe SOAP. Les attachements sont utiles lorsque les données à envoyer risquent de représenter une surcharge inutile pour le processeur SOAP. Imaginez par exemple comment envoyer un fichier son ou image comme partie d'un message SOAP. Ce sont habituellement de gros fichiers et ils ne sont pas encodés en XML. Vous pourriez certainement les inclure dans votre enveloppe SOAP en utilisant un schéma d'encodage binaire comme base64, mais cela impliquerait le chargement et le traitement des données par le moteur SOAP. Cela fonctionnerait certainement mais entraînerait une charge supplémentaire inutile : pourquoi votre processeur SOAP devrait-il traiter un si gros volume de données dont il n'a absolument rien à faire ? Attacher les données permet de les envoyer sans générer de surcharge mais également sans nécessiter de structure XML.

Créons un service d'achat et de vente de voitures d'occasion. Nous aurons une méthode listant les voitures à vendre et récupérant les informations sur les véhicules disponibles. Nous aurons besoin d'un type personnalisé pour représenter une liste. Ce type encapsule les détails de la voiture et les informations permettant de contacter le vendeur. Voici le code de la classe `javasoap.livre.ch8.Liste` que nous allons utiliser pour transmettre cette information. Il comporte une méthode `toString()` afin de pouvoir aisément afficher son contenu.

```
package javasoap.livre.ch8;
public class Liste {
   String _marque;
   String _modele;
   int    _annee;
   int    _kilometres;
   String _nomProprietaire;
   String _emailProprietaire;
   public Liste() {
   }
   public Liste(String marque, String modele,
     int annee, int kilometres, String nomProprietaire,
    String emailProprietaire) {
      setMarque(marque);
      setModele(modele);
      setAnnee(annee);
      setKilometres(kilometres);
      setNomProprietaire(nomProprietaire);
      setEmailProprietaire(emailProprietaire);
   }
   public void setMarque(String marque) {
      _marque = marque;
   }
   public String getMarque() {
      return _marque;
   }
   public void setModele(String modele) {
      _modele = modele;
```

```
    }
    public String getModele() {
       return _modele;
    }
    public void setAnnee(int annee) {
       _annee = annee;
    }
    public int getAnnee() {
       return _annee;
    }
    public void setKilometres(int kilometres) {
       _kilometres = kilometres;
    }
    public int getKilometres() {
       return _kilometres;
    }
    public void setNomProprietaire(String nomProprietaire) {
       _nomProprietaire = nomProprietaire;
    }
    public String getNomProprietaire() {
       return _nomProprietaire;
    }
    public void setEmailProprietaire(String emailProprietaire) {
       _emailProprietaire = emailProprietaire;
    }
    public String getEmailProprietaire() {
       return _emailProprietaire;
    }
    public String toString() {
       String resultat = String.valueOf(_annee) + " " +
                         _marque + " " + _modele +
                         " possédant " + String.valueOf(_kilometres) +
                         " kilomètres:" +
                         "\n  Appartenant à : " + _nomProprietaire + "(" +
                         _emailProprietaire + ")";
       return resultat;
    }
}
```

La classe `Liste` possède les propriétés de type chaîne nommées `_marque` et `_modele`, qui représentent respectivement le constructeur du véhicule et le nom du modèle. L'`_annee` est une propriété de type entier représentant l'année de fabrication du modèle et la propriété de type entier `_kilometres` est le kilométrage de la voiture. `_nomProprietaire` et `_emailProprietaire` sont des propriétés de type chaîne pour les nom et adresse de courrier électronique du propriétaire du véhicule. Les méthodes get et set des propriétés sont conformes au modèle JavaBeans, nous n'avons donc pas à écrire de sérialiseur personnalisé pour la classe `Liste`.

Créons maintenant une classe implémentant le service `urn:ServiceListeVoituresOccasion`. La méthode `ajoutListe()`,[3] qui envoie des listes au service, reçoit en premier paramètre une instance de `Liste` et renvoie un entier représentant l'identifiant de la lis-

te. Nous permettons à l'appelant d'attacher un fichier image à l'invocation de la méthode de service web `ajoutListe()` et le second paramètre est donc une instance de `javax.activation.DataHandler` qui encapsule l'attachement. Apache SOAP possède un sérialiseur pour la classe `DataHandler` et tire parti du JavaBeans Activation Framework (JAF) pour la prise en charge des attachements MIME. Celui-ci rend la réception d'attachements dans un service RPC aussi simple que le passage de paramètres. Le travail de référencement et d'extraction de l'attachement est fait pour vous et vous pouvez manipuler les données au travers de l'API JAF. La méthode `cherche()` possède des paramètres `String` pour la marque et le modèle de la voiture et renvoie un tableau d'identifiants. Enfin, la méthode `getListe()` reçoit en paramètre un identifiant de la liste et renvoie une instance de `Liste` et la méthode `getImage()` reçoit également un identifiant de la liste et renvoie en attachement le fichier image associé. Voici la classe `javasoap.livre.ch8.ServiceListeVoituresOccasion`:

```
package javasoap.livre.ch8;
import java.util.*;
import java.io.*;
import javax.activation.*;
import org.apache.soap.util.mime.*;
public class ServiceListeVoituresOccasion {
   Hashtable _listes = new Hashtable();
   int _IDSuivant = 100;
   public int ajoutListe(Liste liste, DataHandler handler)
               throws Exception {
      _listes.put(new Integer(_IDSuivant), liste);
      int listno = _IDSuivant;
      _IDSuivant++;
      if (handler != null) {
         String nomf = String.valueOf(listno) + ".bin";
         DataSource ds = handler.getDataSource();
         ByteArrayDataSource bsource =
               new ByteArrayDataSource(ds.getInputStream(),
                                       handler.getContentType());
         bsource.writeTo(new FileOutputStream(nomf));
      }
      return listno;
   }
   public Integer[] cherche(String marque, String modele) {
      Vector v = recupIDListes(marque, modele);
      int cnt = v.size();
      if (cnt == 0)
         return null;
      Integer ids[] = new Integer[cnt];
      for (int i = 0; i < cnt; i++) {
         ids[i] = (Integer)v.elementAt(i);
      }
      return ids;
```

3. Naturellement, une `supprimeListe` serait également une bonne idée mais importe peu pour notre exemple.

```
    }
    protected Vector recupIDListes(String marque, String modele) {

       Vector resultat = new Vector();
       for (Enumeration e = _liste.keys();
                            e.hasMoreElements() ;) {
          Integer i = (Integer)e.nextElement();
          Liste liste = (Liste)_liste.get(i);
          if (marque.equals(liste.getMarque()) &&
              modele.equals(liste.getModele())) {
             resultat.add(i);
          }
       }

       return resultat;
    }
    public Liste getListe(int id) {
       Liste liste = (Liste)_liste.get(new Integer(id));
       return liste;
    }
    public DataHandler getImage(int id)
         throws Exception {
       String nomf = String.valueOf(id) + ".bin";
       DataSource ds =
            new ByteArrayDataSource(new File(nomf), null);
       DataHandler dh = new DataHandler(ds);
       return dh;
    }
}
```

Le service opère au niveau application et toutes les listes sont donc disponibles pendant sa durée d'exécution. Nous ne nous sommes pas préoccupés du stockage des listes comme il faudrait bien sûr le faire dans le monde réel. Les fichiers attachés sont toutefois stockés sur disque. Nous allons conserver toutes les listes dans une table de hachage nommée `_listes`, dans laquelle les clés seront les identifiants uniques des listes. Les nouvelles valeurs d'identifiants sont générées par incrémentation de `_IDSuivant` à chaque invocation de la méthode `ajoutListe()`. La première partie d'`ajoutListe()` stocke la liste passée en paramètre dans la table de hachage. La clé est un `Integer` basé sur la valeur de `_IDSuivant` qui est ensuite incrémenté. L'étape suivante consiste à extraire l'attachement et à le stocker localement. Le nom du fichier est le numéro de liste suivi de l'extension *.bin* supprimer. Nous obtenons une référence vers une `javax.activation.DataSource` en appelant la méthode `getDataSource()` du paramètre `handler` puis en créant une `org.apache.soap.util.mime.ByteArrayDataSource`. Le premier paramètre de son constructeur est un `java.io.InputStream` pour les données, que nous obtenons en appelant `ds.getInputStream()`. L'autre paramètre est une `String` contenant le type MIME des données, obtenu en appelant `handler.getContentType()`. Nous pouvons maintenant sauvegarder l'attachement en appelant la méthode `writeTo()` de `ByteArrayDataSource` et en lui passant une instance de `java.io.FileOutputStream` basée sur le nom de fichier que nous avons généré un peu plus tôt. Maintenant que l'attache-

ment a été stocké localement, nous n'avons plus qu'à renvoyer le numéro de liste à l'appelant.

La méthode `cherche()` appelle une méthode locale nommée `recupIDListes()`, qui renvoie un `Vector` contenant des objets `Integer`. Notez que `recupIDListes()` n'est pas exposée en tant que méthode de service. Chaque entrée du `Vector` représente l'identifiant d'une liste correspondant à la marque et au modèle spécifiés. Un `Integer[]` est alors alloué, contenant les valeurs du `Vector` et renvoyé à l'appelant. Ce tableau permet à l'appelant d'utiliser les identifiants pour retrouver les listes réelles ainsi que les images associées.

La méthode `getListe()` reçoit en paramètre un entier, l'identifiant de la liste. Cet identifiant est utilisé pour récupérer l'objet `Liste` depuis la table de hachage `_listes`. La `Liste` est alors renvoyée à l'appelant. La méthode `getImage()` reçoit également l'identifiant de liste sous forme de paramètre entier et l'utilise pour générer le nom de fichier de l'image associée. Ce nom est utilisé pour instancier un `java.io.File`, qui est passé au constructeur de `ByteArrayDataSource`. Il n'est pas nécessaire de spécifier le type des données et on utilise donc `null` comme second paramètre. Un nouveau `DataHandler` est créé à l'aide de la source de données. Il est renvoyé à l'appelant. Vous pouvez maintenant déployer le service en exposant les méthodes de service `ajoutListe`, `cherche`, `getListe` et `getImage`. N'oubliez pas d'ajouter une entrée de mise en correspondance pour le type personnalisé liste, comme ceci :

```
<isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:x="urn:ServiceListeVoituresOccasion" qname="x:Liste"
         javaType="javasoap.livre.ch8.Liste"
java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
```

Passons maintenant à l'application cliente qui envoie une liste au service. Nous n'enverrons pas un gros fichier binaire car nous voulons vérifier que les données sont correctes et nous utiliserons donc un petit fichier lisible nommé *data.bin* :

```
1:Rob
2:Englander
3:MindStream Software, Inc.
```

Voici le code de l'application cliente :

```
package javasoap.livre.ch8;
import java.io.*;
import java.util.*;
import java.net.*;
import org.apache.soap.util.mime.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import javax.activation.*;
import javax.mail.internet.*;
import org.apache.soap.util.xml.*;
public class ClientAjoutListe {
   public static void main(String[] args)
```

```
        throws Exception {
      URL url = new URL(
           "http://localhost:8080/soap/servlet/rpcrouter");
      Call appel = new Call();
      appel.setTargetObjectURI("urn:ServiceListeVoituresOccasion");
      appel.setMethodName("ajoutListe");
      appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      BeanSerializer beanSer = new BeanSerializer();
      appel.setSOAPMappingRegistry(smr);
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
         new QName("urn:ServiceListeVoituresOccasion", "Liste"),
                 javasoap.livre.ch8.Liste.class,
                 beanSer, beanSer);
      Vector params = new Vector();
      Liste liste = new Liste("Pontiac", "Firebird", 1968,
             105000, "Rob Englander", "rob@mindstrm.com");
      params.addElement(new Parameter("liste",
               javasoap.livre.ch8.Liste.class, liste, null));
      DataSource ds = new
             ByteArrayDataSource(new File("data.bin"), null);
      DataHandler dh = new DataHandler(ds);
      params.addElement(new Parameter("handler",
               javax.activation.DataHandler.class, dh, null));
      appel.setParams(params);

      Response rep;
      try {
         rep = appel.invoke(url, "");
      }
      catch (SOAPException e) {
         System.out.println(e.getMessage());
         return;
      }
      if (!rep.generatedFault()) {
         Parameter ret = rep.getReturnValue();
         System.out.println("Le numéro de liste est " +
                                 ret.getValue());
      }
      else {
         Fault faute = rep.getFault();
         System.out.println ("Fault Code = " +
                 faute.getFaultCode());
         System.out.println ("      String = " +
                 faute.getFaultString());
      }
   }
}
```

L'application cliente commence par positionner l'appel qui invoquera la méthode de service `ajoutListe`. Dans la mesure où nous allons envoyer une instance de `Liste`, nous effectuons la mise en correspondance avec une instance de `BeanSerializer`. Une liste est créée, contenant une Pontiac Firebird de 1968 ayant 75.000 kilomètres au compteur et m'appartenant. Ceci est le premier paramètre à l'invocation de la méthode. Une instance de `ByteArrayDataSource` est ensuite créée. Le premier paramètre est un `java.io.File` utilisant le nom de fichier *data.bin*. Une fois encore, nous ne spécifions pas le type de contenu et `null` est donc utilisé en second paramètre. La source de données est alors utilisée pour construire un objet `DataHandler`, qui sert de second paramètre à l'invocation de la méthode.

L'exécution de cet exemple produire le résultat suivant :

```
Le numéro de liste est 100
```

Dans la mesure où le service à une portée de type application, la numérotation des listes s'incrémentera de 1 à chaque exécution du client.

L'enveloppe SOAP envoyée au serveur ne contient pas le fichier image. La couche de transport achemine un message MIME multiparties dans lequel l'enveloppe SOAP constitue la première partie et le fichier attaché la seconde. L'élément XML pour le paramètre handler est donc une référence à la seconde partie et a cet aspect :

```
<handler href ="cid:6889270.1004989967097.apache-soap.localhost"/>
```

À l'exécution de l'exemple, la valeur de la référence sera différente car elle est générée à la volée et inclut le nom de la source (la source est ici `localhost`). Mais ce qui importe est que le contenu du fichier attaché n'apparaît pas au sein de l'enveloppe SOAP. Dans cet exemple, ce contenu est de taille réduite, mais imaginez que nous ayons attaché un fichier son de 10 Mo ou un clip vidéo de la voiture effectuant une course. Si nous descendions au niveau du flux HTTP envoyé au serveur pour cette invocation, nous verrions ceci après la fermeture de l'enveloppe SOAP.

```
Content-Type: application/octet-stream
Content-Transfer-Encoding: 8bit
Content-ID: <6889270.1004989967097.apache-soap.localhost>
Content-Length: 52

1: Rob
2: Englander
3: MindStream Software, Inc.
```

Ceci est le contenu encodé de l'attachement. Dans une vraie application, l'attachement serait probablement un flux conséquent de données binaires, mais dans ce programme, il ne s'agit que d'un court message nous permettant de vérifier que l'attachement a été correctement pris en charge.

Écrivons maintenant un client qui récupèrera les listes et leurs fichiers images associés. Nous allons tout d'abord invoquer la méthode `cherche` en lui envoyant « Pontiac » et « Firebird » en paramètres. En supposant que nous n'avons exécuté `ClientAjoutListe` qu'une seule fois, nous devrions obtenir un tableau entier contenant un seul élément de valeur 100. Il s'agit de l'identifiant de la première et seule liste. Nous utiliserons cet identifiant pour récupérer la liste ainsi que le fichier image (sous forme d'attachement). Voici le code :

```
package javasoap.livre.ch8;
import java.io.*;
import java.util.*;
import java.net.*;
import org.apache.soap.util.mime.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import javax.activation.*;
import javax.mail.internet.*;
import org.apache.soap.util.xml.*;
public class ClientRecherche {
   public static void main(String[] args)
         throws Exception {
      URL url = new URL(
          "http://localhost:8080/soap/servlet/rpcrouter");
      Call appel = new Call();
      appel.setTargetObjectURI("urn:ServiceListeVoituresOccasion");
      appel.setMethodName("cherche");
      appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      BeanSerializer beanSer = new BeanSerializer();
      appel.setSOAPMappingRegistry(smr);
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
          new QName("urn:ServiceListeVoituresOccasion", "Liste"),
          javasoap.livre.ch8.Liste.class, beanSer, beanSer);
      Vector params = new Vector();
      String marque = "Pontiac";
      String modele = "Firebird";
      params.addElement(new Parameter("marque",
                java.lang.String.class, marque, null));
      params.addElement(new Parameter("modele",
                java.lang.String.class, modele, null));
      appel.setParams(params);

      Response rep;
      try {
         rep = appel.invoke(url, "");
      }
      catch (SOAPException e) {
         System.out.println(e.getMessage());
         return;
      }
      if (!rep.generatedFault()) {
         Parameter ret = rep.getReturnValue();
         int[] ids = (int[])ret.getValue();
         int cnt = ids.length;
         for (int i = 0; i < cnt; i++) {
            appel.setMethodName("getListe");
```

```
                params = new Vector();
                params.addElement(new Parameter("id",
                    java.lang.Integer.class, new Integer(ids[i]),
                    null));
                appel.setParams(params);
                try {
                   rep = appel.invoke(url, "");
                }
                catch (SOAPException e) {
                   System.out.println(e.getMessage());
                   return;
                }
                ret = rep.getReturnValue();
                Liste liste = (Liste)ret.getValue();
                System.out.println(liste);
                appel.setMethodName("getImage");
                try {
                   rep = appel.invoke(url, "");
                }
                catch (SOAPException e) {
                   System.out.println(e.getMessage());
                   return;
                }
                ret = rep.getReturnValue();
                DataHandler handler = (DataHandler)ret.getValue();
                DataSource ds = handler.getDataSource();
                String nomf = "EX17." +
                         String.valueOf(ids[i]) + ".bin";
                ByteArrayDataSource bsource =
                      new ByteArrayDataSource(ds.getInputStream(),
                                              handler.getContentType());
                bsource.writeTo(new FileOutputStream(nomf));
                System.out.println(
                      "   Fichier image : " + nomf);
             }
          }
          else {
             Fault faute = rep.getFault();
             System.out.println ("Code faute = " +
                    faute.getFaultCode());
             System.out.println ("      Libellé = " +
                    faute.getFaultString());
          }
       }
    }
```

La première partie du code doit maintenant vous être vraiment familière. Nous préparons un appel à la méthode cherche, mettons en correspondance le type `Liste` et ajoutons deux paramètres chaîne pour la recherche des listes de Pontiac Firebirds. La valeur de retour de l'appel est transtypée en un `int[]`. (Même si la classe du service renvoie un

Integer[], Apache SOAP fait correspondre un tableau d'éléments primitifs avec un tableau correspondant de types Java primitifs et nous obtenons donc un int[].) Nous pouvons maintenant boucler au sein du tableau des identifiants de liste. Pour chacun d'entre eux, nous positionnons le nom de méthode à getListe et utilisons l'identifiant de liste du tableau ids pour créer un Parameter de l'appel. Nous invoquons maintenant la méthode, en transtypant cette fois le résultat en une instance de Liste. Dans la mesure où Liste implémente une méthode toString(), nous pouvons simplement envoyer l'objet liste à System.out.println() pour en afficher son contenu. Nous positionnons ensuite le nom de méthode à getImage() et l'invoquons, en transtypant cette fois la valeur de retour en un DataHandler et en récupérant son DataSource. Nous écrivons le contenu de l'attachement dans un fichier local en utilisant la même technique que pour la classe ServiceListeVoituresOccasion. Le nom de fichier est de la forme *EX17.xxx.bin*, où *xxx* est l'identifiant de la liste.

La dernière étape consiste à afficher le nom du fichier attaché. Le résultat est :

```
1968 Pontiac Firebird possédant 75000 kilomètres :
   Appartenant à Rob Englander(rob@mindstrm.com)
   Fichier image : EX17.100.bin
```

Si vous examinez le contenu du fichier *EX17.100.bin*, vous verrez qu'il est identique à l'original attachement :

```
1: Rob
2: Englander
3: MindStream Software, Inc.
```

Si vous voulez utiliser les attachements dans les services Apache SOAP de type messages, vous devrez travailler à un niveau plus bas, sans les élégantes abstractions fournies au niveau RPC de l'API. GLUE supporte également les attachements SOAP grâce à une API simple à mettre en œuvre. Bien que nous ne les traitions pas ici, je suis certain que vous trouverez les abstractions GLUE simples à comprendre et à utiliser.

Les mécanismes alternatifs SOAP dont nous avons parlé dans ce chapitre, utilisés avec toutes les autres techniques déjà traitées, vous permettent d'utiliser SOAP pour à peu près tout type de transfert de données.

9

Interopérabilité et WSDL

Tous les exemples que nous avons vu jusqu'à présent utilisaient la même implémentation SOAP pour le client et le serveur. Nos exemples de serveurs Apache SOAP étaient accédés par des applications clientes Apache SOAP de même que nos serveurs GLUE l'étaient par des clients GLUE. Dans les rares occasions où vous exercez un contrôle sur chaque nœud d'un système distribué, il est plus facile d'utiliser les mêmes technologies partout. Cette opportunité n'est toutefois pas la règle et elle est par ailleurs fondamentalement contraire à la vision des services web. Il est donc nécessaire de se préoccuper de la façon dont les implémentations SOAP interopèrent les unes avec les autres.

Il existe des douzaines d'implémentations SOAP et d'autres apparaissent régulièrement. Au fil du temps, vous rencontrerez probablement de nombreux systèmes d'entreprise accessibles *via* des mécanismes SOAP et par le biais de diverses couches de transport. Certains de ces systèmes utiliseront SOAP de façon masquée et vous n'aurez donc pas à vous en préoccuper. De plus, les nouvelles structures de logiciels distribués basées sur SOAP pousseront certainement comme des champignons. Comment espérer que ces systèmes interopèreront les uns avec les autres ? Après tout, les logiciels sont principalement développés par des individus et des compagnies en concurrence les uns avec les autres ce qui, en général, pose quelques problèmes en terme d'interopérabilité. Les problèmes surviennent parfois parce que les spécifications sont sujetes à interprétation et que des développeurs travaillant pour des organisations différentes voient les choses différemment. Le fait que ces problèmes existent ne signifie pas pour autant que les spécifications soient défectueuses. C'est simplement le signe que le développement de logiciels est une entreprise humaine et que les humains sont enclins au désaccord.

Une source fréquente de problèmes d'interopérabilité est le champ `SOAPAction`, utilisé par SOAP pour envoyer à un destinataire des informations situées hors des bornes de l'enveloppe SOAP. Ce champ est acheminé comme une partie de l'information d'en-tête de la couche de transport sous-jacente. Par exemple, lorsque le message SOAP est transporté par http, ce champ est envoyé en tant qu'en-tête HTTP. (La façon dont `SOAPAction` doit être pris en charge par les autres couches de transport n'est pas claire.) Le contenu du champ est un URI destiné à fournir des informations au destinataire. Ces informations peuvent être l'espace de noms du service, son nom de méthode ou autre chose. Quoi qu'il en soit, l'intérêt est de fournir au destinataire quelques indications sur le con-

tenu de l'enveloppe sans l'obliger à analyser le XML. Certaines implémentations SOAP n'exigent pas qu'une valeur soit envoyée dans le champ `SOAPAction`, d'autres oui. Savoir ce qui est requis et s'assurer de l'inclure dans le message est critique pour l'interopérabilité SOAP. Nous verrons dans les exemples de ce chapitre les exigences de certaines implémentations concernant `SOAPAction` et les API permettant de lui affecter une valeur. Mais tout d'abord, nous allons nous pencher sur *Web Services Definition Language* (WSDL), un standard important qui promet de régler les problèmes d'interopérabilité.

WSDL

Une manière d'éviter les problèmes d'interopérabilité avec les services SOAP est d'utiliser un langage structuré pour décrire le service, son emplacement, ses méthodes, paramètres et types de données et ainsi de suite. *Web Services Definition Language* (WSDL) sert justement à cela. WSDL est une grammaire XML décrivant les services web. Les systèmes peuvent déterminer les interfaces de programmation d'un service web en examinant le document WSDL associé à ce service. Le document décrit les méthodes du service ainsi que leurs paramètres et valeurs de retour et peut également inclure l'adresse ou destination finale du service. L'un des plus grands avantages de WSDL est qu'il est un standard unique et accepté[1] pour la description de services web ce qui incite, entre autre, les développeurs SOAP à éviter d'utiliser leurs propres mécanismes pour cela.

Est-ce que WSDL élimine pour autant les problèmes associés à l'interprétation humaine (ou de la machine) ? Pas totalement. Il permet aux descriptions de services d'être moins ambiguës que s'il n'existait pas de standard, mais d'un autre côté, WSDL possède ses propres spécifications et celles-ci peuvent potentiellement être interprétées différemment. WSDL n'est pas l'idéal, mais il améliore toutefois le quotidien de ceux qui emploient ces technologies.

Vue d'ensemble de WSDL

Nous n'allons pas couvrir WSDL dans les détails car c'est un vaste sujet qui justifierait un livre entier. Mais il représente une contribution majeure dans le domaine de l'interopérabilité SOAP et il est donc important de lui consacrer un peu de temps. Certaines implémentations SOAP comme GLUE incorporent intégralement WSDL. Les exemples GLUE de ce livre utilisent l'outil `wsdl2java` pour générer le code Java client destiné à accéder à un service GLUE. `wsdl2java` lit un document WSDL décrivant le service afin de créer l'interface Java et les classes. Nous n'avons pas eu à écrire le document WSDL dans ces exemples car GLUE le génère automatiquement. Le processus de liaison du client GLUE utilise également WSDL pour créer le lien dissimulé derrière l'interface Java utilisée pour interagir avec le service.

D'après les spécifications WSDL :

> WSDL est un format XML pour la description de services réseau vus comme un ensemble de points de destination traitant des messages contenant de l'information de type documentaire ou procédurale. Les opérations et les messages sont décrits de façon abstraite puis attachés à un protocole réseau concret et à un format de message pour définir un point de destination. Les points de destination concrets apparentés sont combinés en points abstraits (services).

1. Vous pourrez trouver les spécifications WSDL à *http://www.w3.org/TR/wsdl*.

Un document WSDL est un ensemble d'une ou plusieurs définitions de services. Le document contient un élément racine XML nommé `definitions`. Cet élément contient les définitions du service. L'élément `definitions` peut également contenir un attribut optionnel `targetNamespace`, qui spécifie l'URI associé aux définitions du service. WSDL utilise les espaces de noms et leurs identifiants de la façon dont nous les avons utilisés dans les enveloppes SOAP et il est donc courant de trouver plusieurs espaces de noms déclarés au niveau des `definitions` du document.

WSDL définit les services comme des collections de points de destination sur le réseau. Ces points de destination dans la terminologie WSDL, sont designés comme des `ports`. WSDL separe les `ports` du service de leurs `messages` associés à partir du protocole réseau auquel le service est lié (`liaison`). La combinaison d'une `liaison` et d'une adresse réseau est un `port` et un `service` est une collection de ces `ports`.

Les spécifications WSDL définissent les sections du document principal comme suit :

types

Un conteneur pour des définitions de types de données utilisant un système de types (tel que XSD).

message

Une définition abstraite typée des données communiquées.

operation

Une description abstraite d'une action supportée par le service.

portType

Un ensemble abstrait d'opérations supportées par un ou plusieurs points de destination.

binding

Une spécification de protocole concret et de format de données pour un type de port particulier.

port

Un point de destination unique défini comme la combinaison d'une liaison et d'une adresse réseau.

service

Une collection de points de destination apparentés.

Examinons un service fournissant des cours d'actions en différé ainsi que des titres de nouvelles à partir du symbole d'une action. Le service est nommé `urn:CorpDataServices`. Il possède une méthode nommée `getHeadLines` recevant en paramètre une `String` contenant le symbole de l'action et renvoyant une `String[]` contenant des titres de nouvelles sur une société représentée par un symbole d'action. Le service contient également une méthode nommée `getCotation` recevant le symbole de l'action en paramètre. La méthode renvoie une instance d'un type personnalisé nommé `Cotation` contenant le symbole de l'action, le dernier cours d'échange, l'écart par rapport au cours d'ouverture, le nombre d'actions échangées durant la journée et la date de l'information renvoyée.

Le code suivant implémente le type personnalisé Cotation. J'ai placé cette classe dans le paquetage services. Je préfère séparer les classes serveur des classes client et éviter tout mélange. Un autre paquetage nommé clients sera réservé aux classes clientes. Nous utiliserons les techniques abordées jusqu'à présent pour mettre en correspondance tous les types personnalisés à utiliser dans le code client.

```
package javasoap.livre.ch9.services;
import java.io.*;

import java.util.*;
import java.net.*;
public class Cotation {
   String _symbole;
   float  _dernierPrix;
   float  _change;
   String _derniereModif;
   long   _volume;
   public String getSymbol() {
      return _symbole;
   }
   public void setSymbol(String symbole) {
      _symbole = symbole;
   }
   public float getDernierPrix() {
      return _dernierPrix;
   }
   public void setDernierPrix(float dernierPrix) {
      _dernierPrix = dernierPrix;
   }
   public float getChange() {
      return _change;
   }
   public void setChange(float change) {
      _change = change;
   }
   public String getDerniereModif() {
      return _derniereModif;
   }
   public void setDerniereModif(String derniereModif) {
      _derniereModif = derniereModif;
   }
   public long getVolume() {
      return _volume;
   }
   public void setVolume(long volume) {
      _volume = volume;
   }
}
```

C'est le seul type personnalisé dont nous aurons besoin pour l'instant. Le reste des données est constitué de types standard. Il nous faut ensuite une classe Java pour implémenter le service :

```
package javasoap.livre.ch9.services;
import java.util.Vector;
public class CorpDataService {
   public Cotation getCotation(String symbole)
            throws Exception {
      CotationParser qp = new CotationParser();
      return qp.getCotation(symbole);
   }
   public String[] getHeadLines(String symbole)
            throws Exception {
      HeadlineParser hp = new HeadlineParser();
      Vector v = hp.getHeadLines(symbole);
      int len = v.size();
      String[] result = new String[len];
      for (int i = 0; i < len; i++) {
         result[i] = (String)v.elementAt(i);
      }
      return result;
   }
}
```

La méthode `getCotation()` de notre classe `CorpDataService` utilise une classe nommée `CotationParser`. Cette classe analyse le flux de données du service web de cotation boursière et empaquete le résultat dans une instance de `javasoap.livre.ch9.services.Cotation`. Je ne montrerai toutefois pas le code de cette classe car il est spécifique au système de cotation que j'utilise. Ce service est disponible sur *http://www.mindstrm.com/soap.htm* et vous n'aurez donc pas à l'exécuter vous-même. (Je vous expliquerai comment accéder à ce service un peu plus tard.) De façon similaire, la méthode `getHeadLines()` utilise une classe nommée `HeadlineParser` qui collecte les titres de nouvelles sous forme de `String` et les renvoie dans une instance de `java.util.Vector`. `getHeadLines()` crée une `String[]` basée sur le contenu du vecteur et renvoie le tableau à l'appelant.

J'ai décidé d'implémenter ce service à l'aide de GLUE. Il est situé à *http://mindstrm.com:8004/glue/urn:CorpDataServices*.[2] Voici la `javasoap.livre.ch9.services.CorpDataServices` classe utilisée pour le publier :

```
package javasoap.livre.ch9.services;
import electric.util.Context;
import electric.server.http.HTTP;
import electric.registry.Registry;
public class CorpDataServices {

   public static void main( String[] args )
               throws Exception {
```

2. Assurez-vous d'utiliser *mindstrm.com* et non *www.mindstrm.com*. Ces URL ne pointent pas vers la même adresse.

```
        String ns = "urn:CorpDataServices";
        HTTP.startup("http://mindstrm.com:8004/glue");
        Context context = new Context();
        context.addProperty("activation", "application");
        context.addProperty("namespace", ns);
        context.addProperty("description", "Corporate Services Demo");
        Registry.publish(ns,
           Javasoap.livre.ch9.services.CorpDataService.class, context);
    }
}
```

Une fois que vous aurez exécuté l'exemple CorpDataServices, vous pourrez rechercher le document WSDL associé au services publiés. Nous l'avons fait pour pratiquement tous les exemples GLUE. Le processus de liaison requiert le WSDL qui est généré automatiquement par le serveur. Naturellement, vous pouvez générer votre WSDL à la main. Je préfère laisser le système le faire pour moi. Le moyen le plus simple d'examiner le WSDL d'un service GLUE est un navigateur web. Le WSDL du service urn:CorpDataServices se trouve sur *http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl*. Certains navigateurs, comme Internet Explorer, affichent le WSDL d'un façon qui permet d'étendre les différents nœuds afin de voir leurs contenus. Quel que soit votre navigateur, le WSDL renvoyé par notre application sera le même :

```
<definitions name="CorpDataService"
  targetNamespace="http://www.themindelectric.com/wsdl/CorpDataService/"
  xmlns:tns="http://www.themindelectric.com/wsdl/CorpDataService/"
  xmlns:electric="http://www.themindelectric.com/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.themindelectric.com/package/java.lang/"
  targetNamespace="http://www.themindelectric.com/package/java.lang/">
    <complexType name="ArrayOfstring">
      <complexContent>

        <restriction base="soapenc:Array">
          <attribute ref="soapenc:arrayType" wsdl:arrayType="string[]" />
        </restriction>
      </complexContent>
    </complexType>
  </schema>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns=
     "http://www.themindelectric.com/package/javasoap.livre.ch9.services/"
    targetNamespace=
     "http://www.themindelectric.com/package/javasoap.livre.ch9.services/">
```

```
      <complexType name="Cotation">
        <sequence>
          <element name="_symbole" nillable="true" type="string" />
          <element name="_dernierPrix" type="float" />
          <element name="_change" type="float" />
          <element name="_derniereModif" nillable="true" type="string" />
          <element name="_volume" type="long" />
        </sequence>
      </complexType>
    </schema>
  </types>
  <message name="getCotation0SoapIn">
    <part name="arg0" type="xsd:string" />
  </message>
  <message name="getCotation0SoapOut">
    <part name="Result"
      xmlns:ns1=
        "http://www.themindelectric.com/package/javasoap.livre.ch9.services/"
      type="ns1:Cotation" />
  </message>
  <message name="getTitres1SoapIn">
    <part name="arg0" type="xsd:string" />
  </message>
  <message name="getTitres1SoapOut">
    <part name="Result"
      xmlns:ns1="http://www.themindelectric.com/package/java.lang/"
      type="ns1:ArrayOfstring" />
  </message>
  <portType name="CorpDataServiceSoap">
    <operation name="getCotation" parameterOrder="arg0">
      <input name="getCotation0SoapIn" message="tns:getCotation0SoapIn" />
      <output name="getCotation0SoapOut" message="tns:getCotation0SoapOut" />
    </operation>
    <operation name="getHeadLines" parameterOrder="arg0">
      <input name="getTitres1SoapIn"
             message="tns:getTitres1SoapIn" />
      <output name="getTitres1SoapOut"
             message="tns:getTitres1SoapOut" />
    </operation>
  </portType>
  <binding name="CorpDataServiceSoap" type="tns:CorpDataServiceSoap">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="getCotation">
      <soap:operation soapAction="getCotation" style="rpc" />
      <input name="getCotation0SoapIn">
       <soap:body use="encoded" namespace="urn:CorpDataServices"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </input>
      <output name="getCotation0SoapOut">
```

```
      <soap:body use="encoded" namespace="urn:CorpDataServices"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
    <operation name="getHeadLines">
     <soap:operation soapAction="getHeadLines" style="rpc" />
      <input name="getTitres1SoapIn">
       <soap:body use="encoded" namespace="urn:CorpDataServices"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </input>
      <output name="getTitres1SoapOut">
       <soap:body use="encoded" namespace="urn:CorpDataServices"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>
  <service name="CorpDataService">
    <documentation>Corporate Services Demo
    </documentation>
    <port name="CorpDataServiceSoap" binding="tns:CorpDataServiceSoap">
      <soap:address
        location="http://mindstrm.com:8004/glue/urn:CorpDataServices" />
    </port>
  </service>
  </definitions>
```

Examinons le WSDL généré pour le service `urn:CorpDataServices`. Il semble assez touffu si l'on considère que nous n'avons que deux méthodes de service et un seul type de données personnalisé. (Les documents WSDL ont tendance à être longs et difficiles à comprendre même pour des services très simples. C'est pour cela qu'il vaut mieux laisser un outil les générer pour vous.) La *figure 9-1* illustre la structure de haut niveau du document WSDL de notre service. Le « (n) » après certains éléments indique qu'il existe plus d'un élément de type particulier.

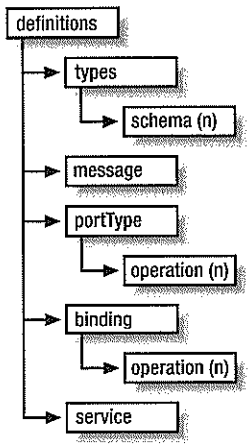


Figure 9-1. Structure de haut niveau d'un document WSDL

L'élément `definitions` constitue la racine du document. Tout le reste se trouve à un niveau inférieur. L'attribut `name` contient le nom des définitions du service, dans notre cas `CorpDataService`. Vous noterez qu'il ne s'agit pas du nom du service, `urn:CorpDataServices`, mais du nom donné aux définitions du service. Ce nom est dérivé du nom de la classe Java qui implémente le service, `javasoap.livre.ch9.services.CorpDataService`. L'attribut `targetNamespace` déclare l'espace de noms associé à la définition du service. Son nom par défaut est créé automatiquement en accolant le nom de la définition à `http://www.themindelectric.com/wsdl`.[3] L'élément `definitions` inclut de nombreuses déclarations d'identifiants d'espaces de noms presque toutes auto-explicatives. Le dernier attribut positionne l'espace de nom global à `http://schemas.xmlsoap.org/wsdl`, qui est l'espace de nom pour WSDL lui-même.

Vient ensuite l'élément `types`. Cet élément définit les types personnalisés utilisés par le service. Chaque type personnalisé est défini à l'aide d'un élément `schema`, où chaque `schema` est un descendant direct des `types`. Le premier schéma est utilisé pour définir le tableau de chaînes renvoyé par la méthode de service `getHeadLines`. Ce tableau est un `complexType`. Souvenez-vous que même un tableau est considéré comme un type complexe en raison de ses valeurs multiples. L'attribut `name` reçoit la valeur `"ArrayOfstring"`. À l'intérieur du `complexType`, on trouve un élément nommé `complexContent`, qui spécifie que le type est restreint à un tableau de chaînes. Le second élément `schema` possède pour l'attribut `name` la valeur `"Cotation"`. Il s'agit du type personnalisé utilisé pour renvoyer les données de cotation depuis le service et il est également défini à l'aide de l'élément `complexType`. Dans ce cas, le `complexType` contient un élément `sequence` qui, à son tour, contient les données du type de données personnalisé cotation. Chaque champ est défini à l'aide d'une balise `d'element` et contient un attribut `name` qui spécifie le nom du champ. GLUE utilise les noms de membres réels de la classe `javasoap.livre.ch9.services.Cotation`. Chaque élément a également son type spécifié.

Les éléments `message` viennent ensuite. Ces éléments définissent l'information transmise entre le client et le serveur. Chaque `message` contient un attribut `name`, formé par le nom de la méthode de service auquel est accolé un nombre qui assure ainsi l'unicité du nom. Enfin, GLUE ajoute `SoapIn` au nom si le message est un paramètre `in`, ou `SoapOut` s'il est un paramètre `out`. Du point de vue de l'élément `message`, il n'existe pas de paramètres `in/out`. Tous les messages sont envoyés vers ou depuis un service.

Chaque `message` contient 0 ou plus éléments `part`. Pour les messages d'entrée d'un service de type RPC tel que le nôtre, chaque `part` représente un paramètre de méthode. Les paramètres `out` du service n'apparaîtront pas dans la description d'un message d'entrée car aucune valeur associée ne sera envoyée par le client vers le serveur. Dans cet exemple, le message nommé `getCotationSoapIn` contient un seul `part` nommé `arg0` de type `string`. Ce `part` correspond au premier (et seul) paramètre de la méthode de service `getCotation`: une chaîne contenant le symbole de l'action. Le message `getCotationSoapOut` contient également un seul `part`, nommé cette fois `Result`. Il s'agit de la valeur de retour de la méthode `getCotation`. Notez que `Result` est défini comme une

3. GLUE utilise `www.themindelectric.com` dans quelques unes des définitions d'espaces de noms qu'il génère dans le WSDL. Vous pouvez supplanter le `targetNamespace` en appelant `context.addProperty()` avec « targetNamespace » comme nom de propriété et la valeur que vous voulez en second paramètre. Toutefois, cela n'éliminera pas l'utilisation de `www.themindelectric.com` dans les autres sections du WSDL.

instance du type Cotation et qu'il est qualifié par un espace de noms à l'aide de http://www.themindelectric.com/package/javasoap.livre.ch9.services/. Ceci fait référence au type personnalisé défini plus haut dans le document WSDL. Les messages getHeadLines suivent le même canevas et nous n'en parlerons donc pas. S'il y avait eu des paramètres out ou in/out renvoyant des valeurs dans n'importe quel de ces messages, ils auraient été définis comme des éléments part immédiatement après le part Result du message xxxSoapOut correspondant. Par exemple, si la méthode getCotation utilisait un paramètre in/out pour le symbole de l'action, l'élément message de la réponse renvoyée aurait ressemblé à ceci :

```
<message name="getCotationOSoapOut">
 <part name="Result"
 xmlns:ns1="http://www.themindelectric.com/package/javasoap.livre.ch9.services/"
    type="ns1:Cotation" />
 <part name="arg0" type="xsd:string" />
</message>
```

L'élément portType est une abstraction définissant un ensemble d'opérations et leurs messages associés. Notre portType possède un attribut name ayant la valeur "CorpDataServiceSoap". Dans ce fichier, le nom est dérivé en ajoutant "Soap" au nom utilisé précédemment dans le document WSDL. Il y a deux éléments operation à l'intérieur du portType, chacun correspondant à une méthode de service. Le premier contient un attribut name possédant la valeur "getCotation", qui est le nom du service. Il contient également un attribut parameterOrder qui spécifie l'ordre dans lequel les paramètres de méthode doivent apparaître. La méthode getCotation possède un seul paramètre, la valeur est donc "arg0". Pour les opérations contenant plus d'un paramètre, la valeur de l'attribut parameterOrder liste chaque paramètre, séparés par des espaces. Chaque operation contient un seul élément input et un seul élément output, chacun contenant un attribut name ainsi qu'un attribut message. name contient le nom du message et message contient le nom qualifié par un espace de noms de l'élément message défini plus haut dans le WSDL.

La section majeure suivante du document WSDL est la section binding. Cette section combine l'abstraction du portType à un protocole physique. L'attribut name de la liaison est, encore une fois, CorpDataServiceSoap. L'attribut type fait référence au type de port implémenté par cette liaison. Dans ce cas, il se réfère à l'élément portType défini la la section WSDL précédente. Sous l'élément binding, on trouve une autre balise binding, celle-ci qualifiée par l'identifiant d'espace de noms soap défini à la racine du document. Cet élément spécifie l'information de liaison SOAP. L'attribut style possède la valeur "rpc", spécifiant qu'il s'agit d'une liaison de service de type RPC. L'attribut transport spécifie la couche de transport sous-jacente. Dans ce cas, la couche de transport est http://schemas.xmlsoap.org/soap/http, qui représente la couche HTTP. Après l'élément soap:binding on trouve les éléments operation pour chacune des opérations du service ou méthodes présentes dans la liaison. Examinons la première, qui possède un attribut name ayant la valeur "getCotation". Il contient un élément nommé soap:operation, possédant deux attributs. L'attribut style possède la valeur "rpc", ce qui signifie que cette liaison représente une invocation SOAP de type RPC. L'attribut soapAction possède la valeur "getCotation", qui est le nom de la méthode elle-même. Il s'agit d'une partie subtile mais importante de l'interopérabilité SOAP. Elle est nécessaire pour envoyer la valeur correcte du champ d'en-tête SOAPAction à l'invocation des services.

C'est un domaine sujet à problèmes avec certaines implémentations SOAP si l'on n'y prend pas garde. Dans cet exemple, l'attribut nous indique que le service veut que le nom de la méthode invoquée soit fourni dans le champ d'en-tête `SOAPAction`. Si vous ne fournissez pas l'en-tête appropriée, cela ne fonctionnera pas. Chaque `operation` possède également un élément `input` et un élément `ouput`. Ces deux éléments partagent la même structure et nous n'examinerons donc que l'`input`. L'élément `input` possède un attribut `name` ayant la valeur `"getCotationOSoapIn"`. Reconnaissez-vous ce nom ? Il s'agit du message défini plus haut par WSDL. Nous associons maintenant le message avec cette opération de liaison. On trouve également un élément `soap:body` spécifiant l'espace de noms de la liaison et le type d'encodage utilisé. Vous pouvez voir que cet exemple utilise un encodage SOAP standard et que l'espace de noms est le nom de service lui-même, `urn:CorpDataServices`. Nous positionnons cet espace de noms dans la classe `CorpDataServices` en appelant la méthode `context.setProperty()`.

Le dernier point de notre exemple WSDL est l'élément `service`. Cet élément combine la liason et une adresse spécifique. L'attribut `name` contient le nom du service qui est dans ce cas `CorpDataService`. Tout cela est suivi d'une balise `documentation` qui contient une brève description textuelle du service. Les données contenues sous cet élément viennent de l'appel de la méthode `context.addProperty("description", "Corporate Services Demo")` que nous avons effectué dans l'application `CorpDataServices` publiant ce service. Si vous ne fournissez pas une description de service, GLUE en dérivera une à partir du nom de la classe Java d'implémentation. Le dernier élément est nommé `port` et il combine la liaison et l'adresse physique. L'attribut `name` est positionné à `"CorpDataServiceSoap"`. Bien qu'il s'agisse juste d'un nom sans réelles implications, j'aime utiliser ce style de notation qui montre immédiatement que l'on parle du service `CorpDataService` utilisant une liaison SOAP. Toutefois, l'attribut `binding` est positionné à `tns:CorpDataServiceSoap`, qui fait référence au `binding` que nous avons vu plus tôt. L'élément `soap:address` contient un attribut `location` contenant l'emplacement réel du service. Pour notre service, l'emplacement est *http://mindstrm.com:8004/glue/urn:CorpDataServices*, l'URL où le service a été publié.

Appeler un service GLUE depuis un client ApacheSOAP

Dans la mesure où nous avons déjà un service GLUE en cours d'exécution, essayons d'y accéder à partir d'un client écrit à l'aide Apache SOAP. Le service est bien aimable de publier du WSDL, mais les clients Apache SOAP n'en ont cure. Nous allons donc devoir dériver nous-même du WSDL toute l'information dont nous avons besoin. Avant cela, écrivons une petite application qui essaiera d'appeler la méthode `getHeadLines` du service `urn:CorpDataServices`.

Nous devons d'abord déterminer la bonne URL. Lorsque nous appelions des services SOAP Apache, l'URL pointait vers la servlet `rpcrouter`. Dans notre cas, une application serveur GLUE héberge le service et l'URL sera donc: *http://mindstrm.com:8004/glue/urn:CorpDataServices*.

Avec de purs exemples Apache SOAP, nous appelions toujours `call.setTargetObjectURI()` et passions le nom de service en paramètre, mais ici nous avons déjà spécifié le nom du service dans l'URL. Il est clair que c'est là le bon choix mais nous devrons appeler `setTargetObjectURI()` avec une valeur quelconque ou nous obtiendrons une er-

reur côté client. GLUE ne semble pas très regardant sur la valeur utilisée dans la mesure où nous sommes parvenus au point de destination final et nous pouvons donc utiliser n'importe quelle chaîne non vide pour l'appel à `setTargetObjectURI()`. J'ai là aussi tendance à utiliser le nom du service mais cela fonctionne avec n'importe quoi. Voici le code de l'application `Apache2Glue` :

```
package javasoap.livre.ch9;
import java.net.*;

import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class Apache2Glue {
   public static void main(String[] args)
      throws Exception {

      URL url =
         new URL("http://mindstrm.com:8004/glue/urn:CorpDataServices");

      Call call = new Call();
      call.setTargetObjectURI("XYZ");

      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = args[0];
      Vector params = new Vector();
      params.addElement(new Parameter("stock", String.class,
                              stock, null));
      call.setParams(params);

      try {
         call.setMethodName("getHeadLines");
         Response resp = call.invoke(url, "");
         if (resp.generatedFault()) {
            Fault fault = resp.getFault();
            String code = fault.getFaultCode();
            String desc = fault.getFaultString();
            System.out.println(code + ": " + desc);
            Vector v = fault.getDetailEntries();
            int cnt = v.size();
            for (int i = 0; i < cnt; i++) {
               Element n = (Element)v.elementAt(i);
               Node nd = n.getFirstChild();
               System.out.println(n.getNodeName() + ": " +
                                          nd.getNodeValue());
            }
         }
         else {
```

```
            Parameter ret = resp.getReturnValue();
            String[] value = (String[])ret.getValue();
            int cnt = value.length;
            for (int i = 0; i < cnt; i++) {
               System.out.println(value[i]);
            }
         }
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                        e.getFaultCode() + "): " +
                        e.getMessage());
      }
   }
}
```

Le reste du code est le même que si nous accédions à un serveur Apache SOAP. Cela n'a pas été très difficile. Il faut juste comprendre quel est l'emplacement du service. Si vous regardez le WSDL de ce service, la réponse sera évidente : l'attribut `location` de l'élément `soap:address` dans la section `service` vous donne l'URL. Le fait que GLUE ne tienne pas compte de l'URI cible est intéressant mais ne devrait pas tellement vous préoccuper.

Si vous exécutez l'application `Apache2Glue` avec un symbole d'action valide en paramètre de la ligne de commande, elle renverra quelques informations relatives à l'action. Par exemple, la commande suivante récupère des nouvelles relatives à General Motors :

```
java javasoap.livre.ch9.clients.Apache2Glue GM
```

Voici le résultat :

```
GMAC files shelf for $8.1 billion in notes
Vivendi Pays $1.5B for EchoStar Slice
Product Is King--Again
GM plans to form venture with SAIC-Wuling in China
Vivendi Pays $1.5B for EchoStar Slice
Daewoo Mtr makes first loan payment in a year-report
'Twas the Night Before Q4'
Valeo Says U.S. Unit Filed for Bankruptcy
Honda's tank in the SUV wars
Daewoo Says to Resume Operations Monday
```

Mon service étant alimenté en temps réel, vous pourrez obtenir des résultats différents lorsque vous exécuterez ce programme.

Maintenant que nous avons un client opérationnel, modifions-le pour appeler la méthode de service `getCotation()`. Dans notre cas, nous allons devoir nous préoccuper d'un type de données personnalisé. Avant d'aller plus loin, nous devrons donc créer une classe `quote` que nous pourrons mettre en correspondance avec la valeur renvoyée par la méthode de service. Le WSDL décrit ce type personnalisé pour nous. Voici les parties significatives du document :

```
<complexType name="Cotation">
  <sequence>
```

```
      <element name="_symbole" nillable="true" type="string" />
      <element name="_dernierPrix" type="float" />
      <element name="_change" type="float" />
      <element name="_derniereModif" nillable="true" type="string" />
      <element name="_volume" type="long" />
    </sequence>
  </complexType>
```

Il n'est pas nécessaire d'écrire un sérialiseur personnalisé pour traiter cela. Nous pouvons écrire un bean Java possédant les méthodes set et get appropriées. Les noms des propriétés et des types de données doivent correspondre aux valeurs des attributs name et type des elements du type Cotation. Par exemple, la classe doit posséder une propriété nommée _symbole et il nous faudra donc des méthodes get_symbole() et set_symbole(). Nous ajouterons également une méthode toString() pour rendre plus aisé l'affichage du contenu de l'objet une fois que nous l'aurons récupéré du service. Voici le code de la classe cliente Cotation :

```
package javasoap.livre.ch9.clients;
import java.io.*;
import java.util.*;
import java.net.*;
public class Cotation {
   String _symbole;
   float  _dernierPrix;
   float  _change;
   String _derniereModif;
   long   _volume;
   public String get_symbole() {
      return _symbole;
   }
   public void set_symbole(String symbole) {
      _symbole = symbole;
   }
   public float get_dernierPrix() {
      return _dernierPrix;
   }
   public void set_dernierPrix(float dernierPrix) {
      _dernierPrix = dernierPrix;
   }
   public float get_change() {
      return _change;
   }
   public void set_change(float change) {
      _change = change;
   }
   public String get_derniereModif() {
      return _derniereModif;
   }
   public void set_derniereModif(String derniereModif) {
      _derniereModif = derniereModif;
   }
```

```
    public long get_volume() {
       return _volume;
    }
    public void set_volume(long volume) {
       _volume = volume;
    }
    public String toString() {
       String result = "Symbole:     " + get_symbole() +
                       "\nDernier Prix: " + get_dernierPrix() +
                       "\nChange:     " + get_change() +
                       "\nDerniere Modif: " + get_derniereModif() +
                       "\nVolume:     " + get_volume();
       return result;
    }
}
```

Nous sommes maintenant prêts à modifier le programme client. J'appelerai la nouvelle classe `Apache2Glue_v2`. Nous devons mettre en correspondance les données renvoyées de `getCotation` avec la classe `Cotation`. Pour cela, il nous faut connaitre l'espace de noms d'où provient le type de données. Encore une fois, le WSDL nous fournit cette information :

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns=
    "http://www.themindelectric.com/package/javasoap.livre.ch9.services/"
  targetNamespace=
    "http://www.themindelectric.com/package/javasoap.livre.ch9.services/">

  <complexType name="Cotation">
    <sequence>
      <element name="_symbole" nillable="true" type="string" />
      <element name="_dernierPrix" type="float" />
      <element name="_change" type="float" />
      <element name="_dernieremodif" nillable="true" type="string" />
      <element name="_volume" type="long" />
    </sequence>
  </complexType>
</schema>
```

Le `targetNamespace` du type personnalisé est défini dans le document WSDL comme étant `http://www.themindelectric.com/package/javasoap.livre.ch9.services/`. Nous utilisons cet espace de noms dans le constructeur `QName` lorsque nous mettons en correspondance les données et notre classe, mais c'est une autre histoire. Le client crée une instance de `SOAPMappingRegistry` et une instance de `BeanSerializer`, tout comme au chapitre 5. Nous utilisons alors la méthode du registre de mise en correspondance `mapTypes()` pour mettre en correspondance l'objet renvoyé par `getCotation` et notre objet `Cotation`. Après cela, nous préparons l'appel de méthode de façon standard, l'effectuons et passons l'objet `Cotation` que nous avons récupéré à `System.out.println()`. Voici le code de `Apache2Glue_v2` :

```
package javasoap.livre.ch9.clients;
import java.net.*;
import java.util.*;
```

```
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class Apache2Glue_v2 {
   public static void main(String[] args)
      throws Exception {

      URL url =
          new URL("http://mindstrm.com:8004/glue/urn:CorpDataServices");
      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      BeanSerializer beanSer = new BeanSerializer();
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
        new QName(
          "http://www.themindelectric.com/package/javasoap.livre.ch9.services/",
          "Cotation"),
          Cotation.class, beanSer, beanSer);

      Call call = new Call();
      call.setSOAPMappingRegistry(smr);
      call.setTargetObjectURI("XYZ");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = args[0];
      Vector params = new Vector();
      params.addElement(new Parameter("stock", String.class,
                                stock, null));
      call.setParams(params);

      try {
         call.setMethodName("getCotation");
         Response resp = call.invoke(url, "");
         if (resp.generatedFault()) {
            Fault fault = resp.getFault();
            String code = fault.getFaultCode();
            String desc = fault.getFaultString();
            System.out.println(code + ": " + desc);
            Vector v = fault.getDetailEntries();
            int cnt = v.size();
            for (int i = 0; i < cnt; i++) {
               Element n = (Element)v.elementAt(i);
               Node nd = n.getFirstChild();
               System.out.println(n.getNodeName() + ": "
                                        + nd.getNodeValue());
            }
         }
         else {
            Parameter ret = resp.getReturnValue();
            Cotation value = (Cotation)ret.getValue();
```

```
                System.out.println(value);
            }
        }
        catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

Voici le résultat :

```
Symbole:      GM
Dernier Prix: 47.28
Change:       -0.2
Derniere Modif: 4:03PM
Volume:       3490100
```

Service d'intermédiation utilisant Apache SOAP

Je vais m'appuyer sur *http://mindstrm.com:8004/glue/urn:CorpDataServices* pour les services que nous allons développer dans le reste de ce chapitre. Dans cet exemple, nous allons créer un service Apache SOAP nommé `urn:ServiceProxyCotation` qui récupèrera ses données du service GLUE que nous avons déjà utilisé. En d'autres termes, ce nouveau service Apache agira en tant qu'intermédiaire avec la partie de récupération de cotation du service GLUE. Ceci est illustré par la *figure 9-2*.

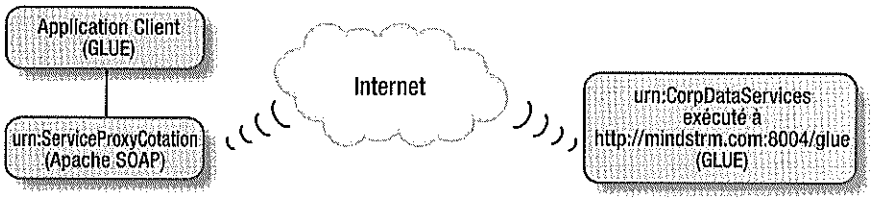


Figure 9-2. Un service d'intermédiation

Au lieu de simplement dupliquer la méthode de service `getCotation()` de `urn:CorpDataServices`, faisons quelque chose de plus intéressant. Après tout, nous sommes libres de concevoir notre service comme nous le voulons. Le fait qu'il récupère ses données depuis un autre service n'est qu'un détail d'implémentation. Enrichissons donc `urn:ServiceProxyCotation` en lui adjoignant une méthode pouvant traiter plusieurs symboles d'actions en un seul appel. Ceci peut être utile pour des applications clientes manipulant plusieurs symboles à la fois. Le nom de la méthode est `getCotations()`. Elle reçoit un tableau de chaînes en paramètre. Chaque élément du tableau est un symbole d'action et la valeur de retour de la méthode est un tableau d'actions. Le nouveau service ne sert pas d'intermédiaire avec la méthode `getHeadLines()` du service `urn:CorpDataServices`. Il n'y a pas de raison particulière, c'est juste un choix de ma part.

Encore une fois, nous allons avoir besoin d'une classe Java qui renverra quelque chose du serveur d'intermédiation. Pour être cohérent avec notre convention de nommage des paquetages, cette classe appartiendra à `javasoap.livre.ch9.services`. Pour éviter toute confusion avec les classes précédentes, appelons celle-ci `ProxyCotation`. Cette classe contient la même information que les autres mais les noms de propriétés et les méthodes des accesseurs sont différents. La seule raison est d'éviter toute confusion. Les noms n'ont pas vraiment d'importance. Voici le code de la classe `ProxyCotation` :

```
package javasoap.livre.ch9.services;
import java.io.*;
import java.util.*;
import java.net.*;
public class ProxyCotation {
   String _sym;
   float  _derniere;
   float  _diff;
   String _date;
   long   _vol;
   public String getStockSymbol() {
      return _sym;
   }
   public void setActionSymbol(String symbole) {
      _sym = symbole;
   }
   public float getDernier() {
      return _dernier;
   }
   public void setDernier(float dernier) {
      _dernier = dernier;
   }
   public float getDiff() {
      return _diff;
   }
   public void setDiff(float diff) {
      _diff = diff;
   }
   public String getDate() {
      return _date;
   }
   public void setDate(String date) {
      _date = date;
   }
   public long getVol() {
      return _vol;
   }
   public void setVol(long vol) {
      _vol = vol;
   }
}
```

Nous avons maintenant une classe Java représentant les données de `urn:ServiceProxyCotation`. Mais n'oublions pas que ce service est également un client de `urn:CorpDataService`. Lorsque nous avons écrit un client Apache SOAP pour ce service, il nous a fallu une classe Java pour effectuer la correspondance avec les données renvoyées. Nous pouvons emprunter cela à `Cotation` : il faut simplement changer le nom du paquetage en `javasoap.livre.ch9.services`, supprimer la méthode `toString()` et changer son nom en `CotationDistante`. Voici le résultat :

```
package javasoap.livre.ch9.services;
import java.io.*;
import java.util.*;
public class CotationDistante {
   String _symbole;
   float  _dernierPrix;
   float  _change;
   String _dernieremodif;
   long   _volume;
   public String get_symbole() {
      return _symbole;
   }
   public void set_symbole(String symbole) {
      _symbole = symbole;
   }
   public float get_dernierPrix() {
      return _dernierPrix;
   }
   public void set_dernierPrix(float dernierPrix) {
      _dernierPrix = dernierPrix;
   }
   public float get_change() {
      return _change;
   }
   public void set_change(float change) {
      _change = change;
   }
   public String get_dernieremodif() {
      return _dernieremodif;
   }
   public void set_dernieremodif(String dernieremodif) {
      _dernieremodif = dernieremodif;
   }
   public long get_volume() {
      return _volume;
   }
   public void set_volume(long volume) {
      _volume = volume;
   }
}
```

Nous avons maintenant toutes les classes de base qu'il nous faut et pouvons avancer dans le code de `urn:ServiceProxyCotation`. La classe Java est nommée `ServiceProxy-`

Cotation et le code est assez simple. Le service expose une méthode, nommée getCotations(), qui récupère un nombre arbitraire de cotations en se basant sur le tableau de symboles d'action. La méthode renvoie une ProxyCotation[], qui est un tableau d'instances de ProxyCotation, une par symbole d'action. Chaque cotation est obtenue par un appel à la méthode getActionCotation(), qui renvoie une simple ProxyCotation basée sur le symbole de l'action passée en paramètre. Chaque cotation est stockée dans un Vector local dont le contenu est utilisé pour remplir un ProxyCotation[] dimensionné de façon appropriée. Ce tableau est alors renvoyé. Le processus utilisé par la méthode getActionCotation() est essentiellement le même que celui de l'application cliente Apache2Glue_v2. La seule différence est que nous ne nous encombrons d'aucun renvoi d'éléments de faute et que nous devons retirer les données de l'instance CotationDistante pour remplir l'instance ProxyCotation. Souvenez-vous que le type de données renvoyé par le service éloigné n'est pas le même que le type de données renvoyé par le nouveau service à ses clients. Et c'est ainsi que cela doit être. Bien qu'il serait plus simple dans cet exemple de renvoyer des instances de CotationDistante, ce n'est pas ainsi que les choses se passent dans le monde réel. Un serveur d'intermédiation ne doit en aucune façon révéler l'existence du serveur sur lequel il s'appuie. Voici le code de ServiceProxyCotation:

```
package javasoap.livre.ch9.services;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class ServiceProxyCotation {
   public ServiceProxyCotation() {
   }
   public ProxyCotation[] getCotations(String[] symboles)
                        throws Exception {
      Vector v = new Vector();
      int cnt = symboles.length;
      for (int i = 0; i < cnt; i++) {
         v.add(getActionCotation(symboles[i]));
      }
      cnt = v.size();
      ProxyCotation[] cotations = new ProxyCotation[cnt];
      for (int i = 0; i < cnt; i++) {
         cotations[i] = (ProxyCotation)v.elementAt(i);
      }
      return cotations;
   }
   ProxyCotation getActionCotation(String symbole)
                     throws Exception {
      URL url = new URL("http://mindstrm.com:8004/glue/urn:CorpDataServices");
      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      BeanSerializer beanSer = new BeanSerializer();
```

```
        smr.mapTypes(Constants.NS_URI_SOAP_ENC,
          new QName(
            "http://www.themindelectric.com/package/javasoap.livre.ch9.services/",
            "Cotation"),

            CotationDistante.class, beanSer, beanSer);

        Call call = new Call();
        call.setSOAPMappingRegistry(smr);
        call.setTargetObjectURI("XYZ");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
        String stock = symbole;
        Vector params = new Vector();
        params.addElement(new Parameter("stock", String.class,
                                  stock, null));
        call.setParams(params);

        call.setMethodName("getCotation");
        Response resp = call.invoke(url, "");
        ProxyCotation quote = new ProxyCotation();
        if (resp.generatedFault()) {
           throw new Exception("Echec de l'appel");
        }
        else {
           Parameter ret = resp.getReturnValue();
           CotationDistante value = (CotationDistante)ret.getValue();
           quote.setActionSymbole(value.get_symbole());
           quote.setDernier(value.get_dernierPrix());
           quote.setDiff(value.get_change());
           quote.setTime(value.get_dernieremodif());
           quote.setVol(value.get_volume());
        }
        return quote;
     }
  }
```

Nous pouvons déployer ce service de la façon dont nous avons déployé tous les services Apache SOAP créés dans ce livre. Nous allons devoir inclure une mise en correspondance pour la classe `ProxyCotation`. Le nom pleinement qualifié du type cotation est `urn:ServiceProxyCotation:Cotation`. Nous utilisons un schéma de nommage des propriétés compatible JavaBeans et la mise en correspondance utilise donc la classe `BeanSerializer` pour les conversions. Voici à quoi ressemble le descripteur de déploiement :

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:ServiceProxyCotation">
  <isd:provider type="java"
                scope="Application"
                methods="getCotations">
    <isd:java class="javasoap.livre.ch9.services.ServiceProxyCotation"
    static="false"/>
  </isd:provider>
  <isd:faultListener>org.apache.soap.server.DOMFaultListener
```

```
    </isd:faultListener>
    <isd:mappings>
      <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:ServiceProxyCotation" qname="x:Cotation"
       javaType="javasoap.livre.ch9.services.ProxyCotation"
       java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
       xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>

    </isd:mappings>
  </isd:service>
```

Appeler un service Apache SOAP depuis un client GLUE

Nous allons cette fois écrire un client GLUE accédant au `urn:ServiceProxyCotation` que nous venons juste de créer. Comment commencer ? GLUE crée les liaisons des services dynamiquement sur la base d'un document WSDL. Dans nos exemples GLUE-vers-GLUE, il générait automatiquement le documents WSDL décrivant le service, mais Apache ne sait faire cela. Nous pourrions certainement écrire un document WSDL approprié à la main... mais c'est un énorme travail et les chances d'obtenir un document correct sont plutôt minces, à moins que vous n'en appreniez sur WSDL plus que quiconque ne doit raisonnablement en savoir. Au lieu de cela, nous allons utiliser l'utilitaire `java2wsdl` fourni avec GLUE pour générer du WSDL.[4] Cet utilitaire se moque pas mal que le code Java ne soit pas destiné à un service GLUE. Il se contente de lire ledit code et de générer le WSDL. `java2wsdl` ne fabrique pas un document WSDL parfait pour un service basé sur Apache mais il est bien plus aisé de le modifier à la main que de l'écrire en entier. Générons donc le WSDL :

```
java2wsdl javasoap.livre.ch9.services.ServiceProxyCotation -n
    urn:ServiceProxyCotation -t urn:ServiceProxyCotation -s
    -e http://localhost:8080/soap/servlet/rpcrouter
```

Le premier paramètre de `java2wsdl` est la classe Java qui implémente le service. L'option `-n` nous permet de spécifier l'espace de noms. Nous utiliserons le nom du service pour cela. L'option `-t` nous permet de spécifier la valeur de l'attribut `targetNamespace`. Ces deux options s'appliquent uniquement aux éléments `definitions` du WSDL généré. Nous aurons besoin de modifier le reste à la main, comme vous le verrez bientôt. L'option `-s` indique à l'utilitaire de générer une liaison SOAP. Nous avons besoin de cette option car nous sommes en train de créer une définition de service pour un point de destination connu (l'emplacement où le service a été déployé) et nous savons par conséquence quelle couche de transport nous utilisons. (Il est plutôt courant de définir l'interface vers le service et ses liaisons séparément, ainsi le même service peut être déployé vers différents points de destination avec des protocoles différents.) Pour cette raison, nous incluons également l'option `-e`, qui nous permet de spécifier ce point de destination. Dans la mesure où nos services Apache SOAP de type RPC sont tous routés par la servlet *rpcrouterr*, le point de destination est *http://localhost:8080/soap/servlet/rpcrouter*.[5]

4. D'autres outils de même utilité sont disponibles à d'autres sources. Par exemple, l'IBM web services toolkit inclut un utilitaire similaire.

Maintenant que nous avons généré un fichier WSDL, effectuons les ajustements nécessaires. Les parties modifiées du fichier apparaissent en gras.

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions
name='urn:ServiceProxyCotation'
targetNamespace='urn:ServiceProxyCotation'
xmlns:tns='urn:ServiceProxyCotation'
xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
xmlns:http='http://schemas.xmlsoap.org/wsdl/http/'
xmlns:mime='http://schemas.xmlsoap.org/wsdl/mime/'
xmlns:xsd='http://www.w3.org/2001/XMLSchema'
xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
xmlns='http://schemas.xmlsoap.org/wsdl/'>
  <types>
    <schema xmlns='http://www.w3.org/2001/XMLSchema'
       xmlns:tns='urn:ServiceProxyCotation'
       targetNamespace='urn:ServiceProxyCotation'>
      <complexType name='Cotation'>
        <sequence>
          <element name='actionSymbole' nillable='true' type='string'/>
          <element name='dernier' type='float'/>
          <element name='diff' type='float'/>
          <element name='date' nillable='true' type='string'/>
          <element name='vol' type='long'/>
        </sequence>
      </complexType>
      <complexType name='ArrayOfCotation'>
        <complexContent>
          <restriction base='soapenc:Array'>
            <attribute ref='soapenc:arrayType'
                 wsdl:arrayType='tns:Cotation[]'/>
          </restriction>
        </complexContent>
      </complexType>
    </schema>
    <schema xmlns='http://www.w3.org/2001/XMLSchema'

       xmlns:tns='urn:ServiceProxyCotation'
       targetNamespace='urn:ServiceProxyCotation'>
      <complexType name='ArrayOfstring'>
        <complexContent>
          <restriction base='soapenc:Array'>
            <attribute ref='soapenc:arrayType' wsdl:arrayType='string[]'/>
          </restriction>
        </complexContent>
```

5. Cet emplacement fait partie de mon réseau local. Vous devrez utiliser votre propre adresse de réseau pour l'emplacement que vous utilisez pour déployer le service.

```
    </complexType>
  </schema>
</types>
<message name='getCotationsOSoapIn'>
  <part name='arg0'
     xmlns:ns1='urn:ServiceProxyCotation' type='ns1:ArrayOfstring'/>
</message>
<message name='getCotationsOSoapOut'>
  <part name='Result'
     xmlns:ns1='urn:ServiceProxyCotation' type='ns1:ArrayOfCotation'/>
</message>
<portType name='ServiceProxyCotationSoap'>
  <operation name='getCotations' parameterOrder='arg0'>
    <input name='getCotationsOSoapIn' message='tns:getCotationsOSoapIn'/>
    <output name='getCotationsOSoapOut' message='tns:getCotationsOSoapOut'/>
  </operation>
</portType>
<binding name='ServiceProxyCotationSoap' type='tns:ServiceProxyCotationSoap'>
  <soap:binding style='rpc'
      transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='getCotations'>
    <soap:operation soapAction='getCotations' style='rpc'/>
    <input name='getCotationsOSoapIn'>
      <soap:body use='encoded' namespace='urn:ServiceProxyCotation'
          encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </input>
    <output name='getCotationsOSoapOut'>
      <soap:body use='encoded' namespace='urn:ServiceProxyCotation'
          encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </output>
  </operation>
</binding>
<service name='ServiceProxyCotation'>
  <port name='ServiceProxyCotationSoap'
        binding='tns:ServiceProxyCotationSoap'>
    <soap:address
        location='http://localhost:8080/soap/servlet/rpcrouter'/>
  </port>
</service>
</definitions>
```

Nous devons modifier les valeurs de l'espace de noms et de l'espace de noms de la cible pour tous les éléments excepté `definitions`. Cet élément a été correctement pris en compte en raison des options `-n` et `-t` spécifiées dans la ligne de commande de lancement de `java2wsdl`. C'est pour cela que vous voyez les valeurs modifiées pour tous les attributs `namespace` et `targetNamespace`. Les valeurs générées contiennent des noms basés sur les classes Java utilisées par l'implémentation du service. Il s'agit juste de noms mais je n'aime pas voir ce genre d'information exposé. Dans d'autres parties du document, j'ai supprimé des noms de paquetages Java similaires qui préfixaient des noms acceptables tels quels. Par exemple, la valeur de l'attribut `name` de l'élément `binding` est

préfixée du nom du paquetage dans lequel se trouvent nos classes de service. Les modifications de l'attribut `name` n'étaient pas nécessaires pour l'interopérabilité, mais je pense qu'elles éliminent toute confusion compte-tenu du besoin d'exposer les classes Java et les noms de paquetages.

Quelques modifications sont nécessaires dans la définition du type personnalisé. Le nom généré pour le type était `ProxyCotation`, qui provient du nom de la classe Java qui implémente le type personnalisé côté serveur. Ce n'est toutefois pas le nom que nous avons choisi dans la section de mise en correspondance du descripteur de déploiement. Examinons de nouveau les parties significatives de ce descripteur :

```
<isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:x="urn:ServiceProxyCotation" qname="x:Cotation"
javaType="javasoap.livre.ch9.services.ProxyCotation"
java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
```

L'espace de noms est `urn:ServiceProxyCotation`. Nous avons déjà modifié la partie correspondante du WSDL et elle est donc correcte. Mais regardez l'attribut `qname`. Nous lui avons attribué la valeur `Cotation`,[6] et non `ProxyCotation`. Cela signifie que lors de l'encodage de l'enveloppe de réponse, les données de cotation seront typées en `Cotation` et qualifiée par l'espace de noms `urn:ServiceProxyCotation`. Le générateur `java2wsdl` n'a aucun moyen d'en être informé car l'information réside hors du code Java.

Enfin, nous devons modifier les noms des éléments de données qui forment le type cotation personnalisé. L'utilitaire `java2wsdl` n'utilise pas par défaut les accesseurs de propriétés JavaBeans. Au lieu de cela, il examine directement les variables des membres dans la classe d'implémentation. Et ainsi, les noms utilisés sont `sym`, `_dernier`, etc. Mais la classe `BeanSerializer` qui encode les instances de la classe `ProxyCotation` utilise les noms de propriété basés sur les conventions de nommage JavaBeans. Enfin, presque. Ils ne sont pas en réalité encodés comme on l'attendrait. Si vous regardez l'enveloppe SOAP renvoyée par le service, vous verrez que les éléments du type personnalisé cotation utilisent une minuscule comme première lettre du nom de propriété. Conformément aux spécifications JavaBeans, les accesseurs en lecture et en écriture nommés `setQuelqueChose()` et `getQuelqueChose()` signifient que le nom de la propriété est `quelqueChose`. Mais le `BeanSerializer` les encode en `quelquechose`. Ce problème ne nous affectait pas avec les exemples Apache-vers-Apache, mais les choses en vont autrement avec les clients GLUE. Nous avons donc quelques petits problèmes introduits par les particularités de chaque implémentation SOAP. Pour les corriger, les noms d'éléments sont modifiés en `actionSymbole`, `dernier`, `diff`, `date` et `vol`.

Il y a également une définition pour le tableau de cotations qui est renvoyé par la méthode de service `getCotations`. La valeur de l'attribut `name` est modifiée en `ArrayOfCotation` et la valeur de `wsdl:arrayType` ets modifiée en `tns:Cotation[]`. En fait, nous supprimons « Proxy » de la définition de la cotation et du tableau associé. La seconde définition de `message` se réfère au tableau de cotations et nécessite donc d'être également modifiée pour refléter le nom du type tableau de cotations.

---

6. La valeur est en fait `x:Cotation`, mais pour plus de clarté, je me réfère seulement à la partie locale du nom plutôt qu'au nom pleinement qualifié.

Nous devons maintenant publier ce document WSDL. Mon processeur Apache SOAP est hébergé par un serveur Apache Tomcat. Ce serveur écoute sur le port 8080. Le répertoire *webapps/soap* sous mon installation Apache SOAP est un endroit parfait pour copier le fichier WSDL. Cela signifie que je peux y accéder par l'URL *http://localhost:8080/soap/ServiceProxyCotation.wsdl*.

À ce stade, nous sommes prêts à écrire un client GLUE pour accéder à `urn:ServiceProxyCotation`. Toutes les classes de cette application doivent être dans le paquetage `javasoap.livre.ch9.clients`. Nous n'allons pas utiliser l'utilitaire `wsdl2java` pour créer les classes et les interfaces pour la liaison à ce service car nous ne pouvons contrôler les noms des classes. Or nous devons avoir ce contrôle car nous mettons toutes les classes côté client dans le paquetage `javasoap.livre.ch9.clients` et nous avons déjà une classe `Cotation`. Nous voulons que la classe Java locale mise en correspondance avec le type de données cotation soit nommée `ProxyCotation`. Vous pouvez exécuter `wsdl2java` depuis un répertoire autre que celui correspondant au paquetage `javasoap.livre.ch9.clients`. Renommez la classe `Cotation` générée ainsi que le fichier en `ProxyCotation`. Modifiez-y les références au fichier *ServiceProxyCotation.map*. Puis déplacez les fichiers dans le répertoire du paquetage. Maintenant, nous allons écrire la classe à la main. Ce n'est pas difficile et cela nous donnera l'opportunité d'examiner les composants nécessaires à une application cliente GLUE.

Créons donc une classe cliente nommée `ProxyCotation` pour représenter les données de cotation renvoyées par le service. Cette classe nécessite un champ public pour chaque élément de la cotation et nous ajouterons une méthode `toString()` pour rendre l'affichage plus aisé. Voici le code de la classe `ProxyCotation` :

```
package javasoap.livre.ch9.clients;
public class ProxyCotation {
   public String actionSymbole;
   public float dernier;
   public float diff;
   public String date;
   public long vol;
   public String toString() {
      String result = "Symbol:      " + actionSymbole +
                      "\nDernier Prix: " + dernier +
                      "\nChange:      " + diff +
                      "\nDerniere Modif: " + date +
                      "\nVolume:      " + vol;
      return result;
   }
}
```

Nous aurons besoin d'un fichier pour décrire la mise en correspondance des données renvoyées par le service vers la classe `ProxyCotation`. Le fichier se nomme *ServiceProxyCotation.map*. Je l'ai placé dans le même répertoire que les classes `javasoap.livre.ch9.clients` afin qu'il soit automatiquement chargé. Le voici :

```
<?xml version='1.0' encoding='UTF-8'?>
<mappings xmlns='http://www.themindelectric.com/schema/'>
  <schema xmlns='http://www.w3.org/2001/XMLSchema'
```

```
        targetNamespace='urn:ServiceProxyCotation'
        xmlns:electric='http://www.themindelectric.com/schema/'>
      <complexType
           name='Cotation'
           electric:class='javasoap.livre.ch9.clients.ProxyCotation'>
        <sequence>
          <element name='actionSymbole' nillable='true'
                   electric:field='actionSymbole' type='string'/>
          <element name='vol'
                   electric:field='vol' type='long'/>
          <element name='dernier'
                   electric:field='dernier' type='float'/>
          <element name='diff'
                   electric:field='diff' type='float'/>
          <element name='date' nillable='true'
                   electric:field='date' type='string'/>
        </sequence>
      </complexType>
    </schema>
  </mappings>
```

Notez que nous avons donné à l'attribut `name` de l'élément `complexType` la valeur `Cotation`. Cela correspond à ce que nous avons utilisé dans le descripteur de déploiement pour le service Apache et à la modification effectuée dans le fichier WSDL. Les attributs `name` de toutes les balises `element` correspondent aux noms des éléments des données de cotation provenant du service et les valeurs de l'attribut `electric:field` correspondent aux noms de variables des membres de la classe `ProxyCotation` mise en correspondance. L'attribut `targetNamespace` qui pilote tout le document de mise en correspondance possède la valeur `urn:ServiceProxyCotation`. C'est la valeur que l'on s'attend à voir utilisée par le service pour qualifier par un espace de noms le type `Cotation` renvoyé. De plus cela correspond également au WSDL.

Voici l'interface Java que nous utiliserons pour nous lier au service. Elle contient la seule et unique méthode de service `getCotations()`.

```
package javasoap.livre.ch9.clients;
public interface IServiceProxyCotation {
  ProxyCotation[] getCotations(String[] symboles);
}
```

Nous avons maintenant tout ce qu'il nous faut pour écrire une application cliente. Nous utiliserons des arguments en ligne de commande pour spécifier les symboles d'action dont nous voulons les cotations. Nous nous lierons au service en utilisant l'adresse du documents WSDL que nous avons publié, en invoquant la méthode `getCotations()` en utilisant `args` en paramètre et en affichant les résultats par itération dans le tableau renvoyé. Voici le code :

```
package javasoap.livre.ch9.clients;
import electric.registry.Registry;
public class Glue2Apache {
   public static void main( String[] args )
            throws Exception {
```

```
        IServiceProxyCotation service = (IServiceProxyCotation)Registry.bind(
              "http://localhost:8080/soap/ServiceProxyCotation.wsdl",

              IServiceProxyCotation.class);
        ProxyCotation[] quotes = service.getCotations(args);
        int cnt = quotes.length;
        for (int i = 0; i < cnt; i++) {
           System.out.println(quotes[i]);
           System.out.println("");
        }
     }
  }
```

J'ai appelé le service en ligne de commande en passant quatre symboles d'actions :

```
java javasoap.livre.ch9.clients.Glue2Apache IBM MSFT GM MMM
```

Voici le résultat obtenu :

```
Symbole:     IBM
Dernier Prix: 123.1
Change:      -0.79
Derniere Modif: 1:59PM
Volume:      3916800
Symbol:      MSFT
Dernier Prix: 67.701
Change:      -1.789
Derniere Modif: 2:04PM
Volume:      21485800
Symbole:     GM
Dernier Prix: 47.36
Change:      -0.74
Derniere Modif: 1:59PM
Volume:      1719700
Symbole:     MMM
Dernier Prix: 118.51
Change:      -1.29

Derniere Modif: 2:00PM
Volume:      953400
```

Accéder à des services .NET

Bien qu'elle ne soit pas basée sur Java, il est impossible de passer sous silence la technologie Microsoft.NET. Si vous interagissez avec des services web, vous rencontrerez certainement plus d'une fois .NET. Pour montrer comment obtenir une interopérabilité entre .NET et les services web écrits en Java, j'ai écrit un service web .NET à l'aide du langage C# et l'ai publié sur Internet à *http://mindstrm.com:8199/CorpDataService/Proxy.asmx*.[7] Pour ceux d'entre vous qui utilisez .NET, il est nécessaire de spécifier que votre service est de type RPC, car il ne s'agit pas du type de service par défaut sous .NET.

7. Le code de ce service est situé à *http://www.mindstrm.com/soap.htm*.

Pour cela, utilisez la déclaration `[SoapRpcService()]` dans votre code C#. Mon service .NET agit comme un intermédiaire avec le service `urn:CorpDataServices` exécuté à *http://mindstrm.com:8004/glue/urn:CorpDataServices*. La *figure 9-3* représente les relations mises en œuvre.

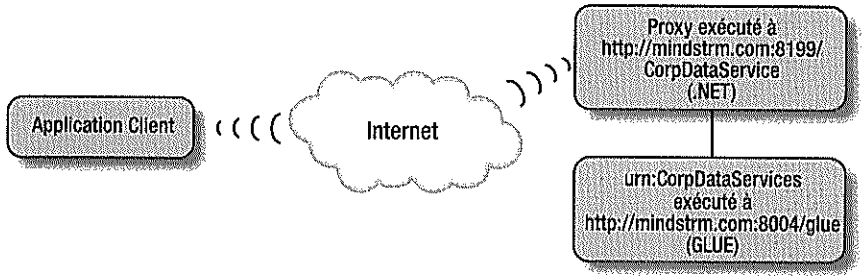


Figure 9-3. Interaction entre un client, un proxy et un service

Le service `Proxy` possède une méthode `getHeadLines` ainsi qu'une méthode `getCotation`, tout comme le service `urn:CorpDataServices`. Chaque méthode reçoit un paramètre de type chaîne contenant un symbole d'action. `getHeadLines` renvoie un tableau de chaînes et `getCotation` renvoie un type cotation personnalisé. Vous pouvez trouver de l'information sur les services .NET en naviguant simplement vers leur URL. Entrez l'URL *http://mindstrm.com:8199/CorpDataService/Proxy.asmx* dans votre navigateur et la page montrée *figure 9-4* s'affichera.

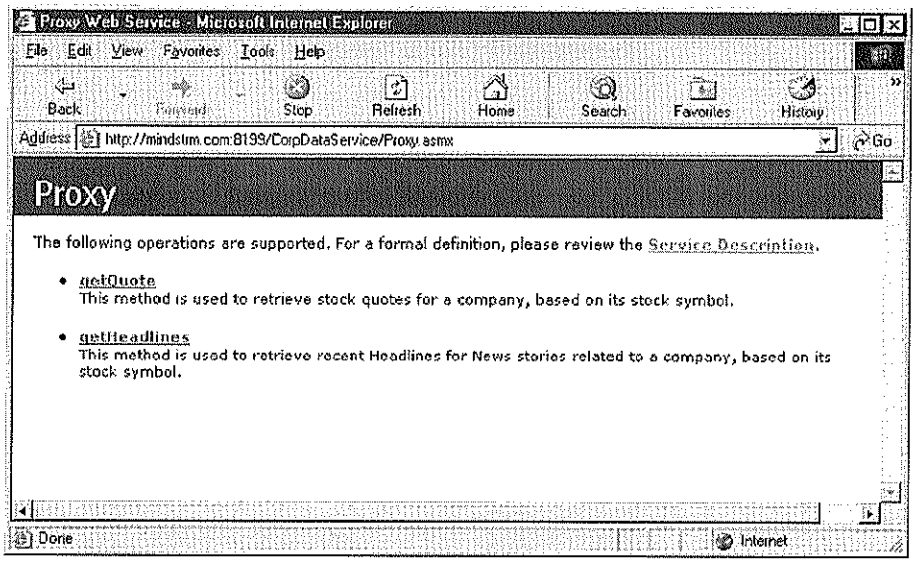


Figure 9-4. Information sur un service .NET

Si vous cliquez sur le lien `getQuote` (pour `getCotation`) ou le lien getHeadLines, vous serez emmené vers une page vous permettant de voir des exemples du XML permettant

d'envoyer des requêtes à la méthode de service ainsi que les réponses associées. Il existe également une interface élémentaire qui vous permet d'utiliser la méthode de service sans écrire le moindre bout de code. Cliquez sur Service Description affiche le WSDL de tout le service `Proxy`. Nous pouvons examiner le WSDL pour y trouver les `SOAPAction` requis et autres espaces de noms et nous pouvons ainsi paramétrer correctement notre Apache SOAP. Il existe toutefois une méthode plus simple. Dans la mesure où cliquer sur le nom d'une méthode affiche une page qui montre le XML des messages transmis entre un client et la méthode de service, nous pouvons utiliser notre navigateur pour nous représenter quelles sortes de messages sont à envoyer. La *figure 9-5* montre la section de cette page et à quoi l'enveloppe de la requête doit ressembler pour invoquer `get-Cotation`.

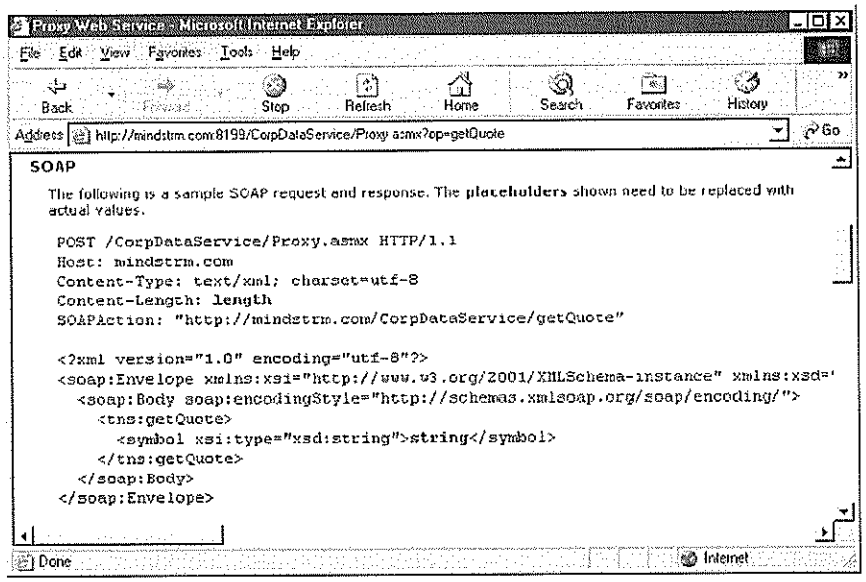


Figure 9-5. Le document SOAP envoyé

Ceci nous donne à peu près tout ce dont nous avons besoin pour accéder au service, sauf son adresse. Bien que le champ `Host` de l'en-tête HTTP montre que le nom du serveur est `mindstrm.com`, elle ne dit pas quel port est à utiliser pour y accéder. Pour obtenir l'adresse correcte, nous devons jeter un coup d'œil au WSDL. Voici la section `service` du document WSDL :

```
<service name="Proxy">
  <port name="ProxySoap" binding="s2:ProxySoap">
  <soap:address
    location="http://mindstrm.com:8199/CorpDataService/Proxy.asmx" />
  </port>
```

Nous voyons maintenant que l'URL est *http://mindstrm.com:8199/CorpDataService/Proxy.asmx*. Nous avons ensuite besoin de la valeur du champ d'en-tête `SOAPAction`. Eh oui, dans ce cas, c'est important ! Si vous ne faites pas cela, .NET n'invoquera pas votre service. Nous devons donc spécifier la `SOAPAction` dans notre application, à l'aide de la

valeur `http://mindstrm.com/CorpDataService/getCotation`. Souvenez-vous, ceci n'est pas une adresse à utiliser pour l'emplacement du service ou de la méthode de service et n'inclut donc pas de détails tels qu'un numéro de port. C'est un URI, un nom qui indique à la structure .NET de quoi traite l'enveloppe SOAP.

Nous devons maintenant déterminer l'espace de noms utilisé pour qualifier l'élément `getCotation`. `getCotation` est qualifié à l'aide de l'identifiant d'espace de noms `tns`. (Cela ne se voit pas dans la *figure 9-5*, mais vous le verrez si vous faites défiler la page vers la droite.) Voici la déclaration d'espace de noms :

```
xmlns:tns=http://mindstrm.com/CorpDataService
```

Cette déclaration nous indique que l'espace de noms voulu est `http://mindstrm.com/CorpDataService`. La dernière chose à noter est le nom du paramètre transmis à la méthode `getCotation`. Son nom est `symbole` et il a de l'importance pour .NET.

Avant d'écrire du code, ne devons examiner la réponse que le service renverra. Cette réponse nous dit tout ce que nous devons savoir sur le type de données personnalisé cotation que nous avons à traiter. La *figure 9-6* montre un exemple de l'enveloppe de la réponse.

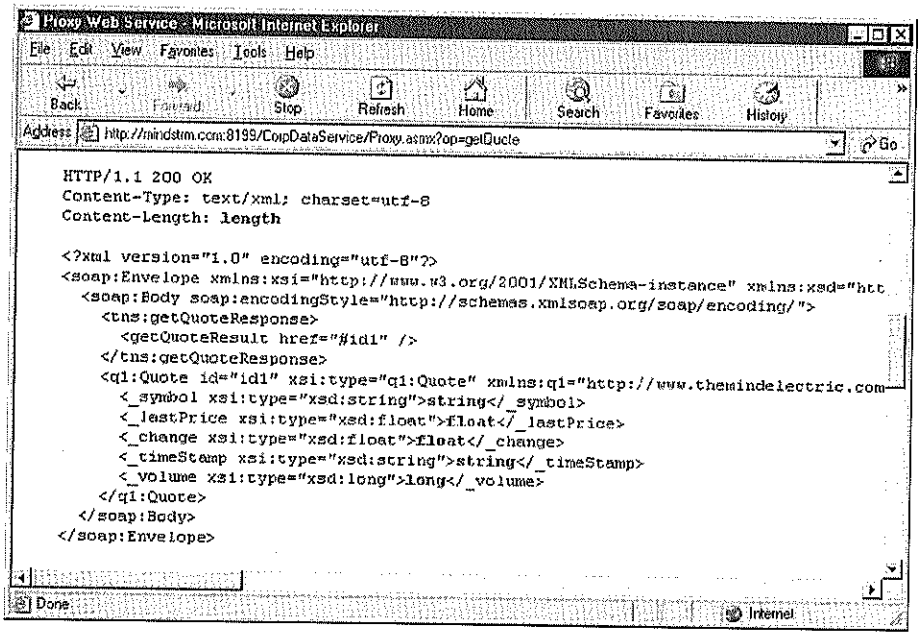


Figure 9-6. Réponse à un appel

`getCotationResponse` est qualifiée par `tns`, qui possède la même valeur que dans la requête : `http://mindstrm.com/CorpDataService`. (L'espace de noms n'est pas dans la partie visible de la page.) Notez également que l'élément `Cotation` est également qualifié par un espace de noms, à l'aide de l'identifiant `q1`. Encore une fois, l'espace de nom est tronqué à l'affichage et voici ce qu'il contient :

```
xmlns:q1="http://www.themindelectric.com/package/javasoap.livre.ch9.services/"
```

Que fait `www.themindelectric.com` dans ce XML ? Lorsque j'ai développé le service .NET, j'ai utilisé Visual Studio .Net pour générer une classe C# qui accède à `urn:CorpDataServices` implémenté en GLUE. Il a procédé en réclamant le WSDL généré automatiquement par le serveur GLUE. Ce serveur utilise `www.themindelectric.com` et le nom du paquetage Java lorsqu'il crée l'espace de noms. Souvenez-vous qu'il s'agit juste d'un nom. Le connaître sera utile lorsque nous mettrons en correspondance ce type avec une classe Java.

Voici le code de l'application cliente Apache SOAP accédant au service .NET :

```
package javasoap.livre.ch9.clients;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;

public class Apache2DotNet {

   public static void main(String[] args)
      throws Exception {

      URL url =
           new URL("http://mindstrm.com:8199/CorpDataService/Proxy.asmx");
      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      BeanSerializer beanSer = new BeanSerializer();
      smr.mapTypes(Constants.NS_URI_SOAP_ENC, new QName(
        "http://www.themindelectric.com/package/javasoap.livre.ch9.services/",
        "Cotation"),
         Cotation.class, beanSer, beanSer);

      Call call = new Call();
      call.setSOAPMappingRegistry(smr);
      call.setTargetObjectURI("http://mindstrm.com/CorpDataService");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = args[0];
      Vector params = new Vector();
      params.addElement(new Parameter("symbole",
                              String.class, stock, null));
      call.setParams(params);

      try {
         call.setMethodName("getCotation");
         Response resp = call.invoke(url,
               "http://mindstrm.com/CorpDataService/getCotation");
         if (resp.generatedFault()) {
            Fault fault = resp.getFault();
            String code = fault.getFaultCode();
```

```
                String desc = fault.getFaultString();
                System.out.println(code + ": " + desc);
                Vector v = fault.getDetailEntries();
                int cnt = v.size();
                for (int i = 0; i < cnt; i++) {
                   Element n = (Element)v.elementAt(i);
                   Node nd = n.getFirstChild();
                   System.out.println(n.getNodeName() + ": " +
                                                   nd.getNodeValue());
                }
             }
             else {
                Parameter ret = resp.getReturnValue();
                Cotation value = (Cotation)ret.getValue();
                System.out.println(value);
             }
          }
          catch (SOAPException e) {
             System.err.println("Lève une SOAPException (" +
                           e.getFaultCode() + "): " +
                           e.getMessage());
          }
       }
    }
```

La classe Java `Apache2DotNet` est similaire à la classe `Apache2Glue_v2` développée plus haut, nous allons donc juste examiner les modifications et non la classe toute entière. Les modifications apparaissent en gras pour que vous puissiez les repérer facilement.

L'URL nécessaire pour accéder au service est *http://mindstrm.com:8199/CorpDataService/Proxy.asmx*. Dans la mesure où nous allons invoquer `getCotation`, nous allons récupérer une instance du type personnalisé cotation. Vous aurez peut-être remarqué dans la *figure 9-6* que les champs de ce type cotation sont exactement les mêmes que ceux du type `Cotation` renvoyé par `urn:CorpDataServices`. Nous pouvons donc réutiliser ici du code par une mise en correspondance avec la classe `Cotation` développée lorsque nous avons écrit le client Apache SOAP accédant au service GLUE. Il se trouve que l'espace de noms dont nous avons besoin pour qualifier le type cotation est également le même : `http://www.themindelectric.com/package/javasoap.livre.ch9.services`.

La modification suivante concerne le paramètre envoyé à la méthode `call.setTargetObjectURI()`. Nous envoyons dans ce cas `http://mindstrm.com/CorpDataService`, qui est l'espace de noms utilisé pour qualifier l'élément `getCotation` dans le WSDL. Nous avons également besoin de modifier le nom du paramètre spécifié dans le constructeur `Parameter()` en `symbole`, puisque c'est le nom que nous avons trouvé dans l'enveloppe de la requête.

La dernière modification consiste à spécifier la valeur de `SOAPAction` associée à l'invocation de cette méthode. Pour positionner cette valeur, nous utilisons le second paramètre de la méthode `call.invoke()`. Dans nos autres clients Apache SOAP, nous passions une chaîne vide en second paramètre car nous devions spécifier une `SOAPAction`. Toutefois, lorsque nous invoquons les méthodes d'un service .NET, nous devons inclure la

SOAPAction appropriée. Nous avons vu plus haut que SOAPAction devait avoir pour valeur http://mindstrm.com/CorpDataService/getCotation, c'est donc ce que nous envoyons à call.invoke().

Le reste du code ne change pas. Si vous exécutez l'application Apache2DotNet, elle invoquera la méthode getCotation de notre service .NET, qui à son tour invoque la méthode getCotation du service urn:CorpDataServices. Voici à quoi ressemble le résultat lorsque j'exécute l'application en lui passant le symbole d'action GM :

```
Symbole:      GM
Dernier Prix: 48.05
Change:       0.35
Derniere Modif: 11:06AM
Volume:       1078700
```

Écrire un client Apache Axis

Le groupe Apache travaille actuellement sur une implémentation SOAP totalement nouvelle connue sous le nom d'Axis.[8] Il est vraiment trop tôt pour y passer beaucoup de temps. Elle sera sujète à modifications avant de se stabiliser et n'est pas prête pour un usage en production. Il est prévu qu'Axis se conforme à JAX-RPC, un standard d'API émergeant pour le RPC SOAP. Cependant, Axis est disponible et peut être utilisé à des fins d'expérimentation. Nous allons donc y jeter un coup d'œil et voir comment il fonctionne. Plus loin, au chapitre 11, nous aborderons JAX-RPC.

Le cœur du client est Axis2DotNet. Elle invoque la méthode getHeadLines du service .NET que nous avons utilisé précédemment.

```
package javasoap.livre.ch9.clients;
import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.client.Transport;

import org.apache.axis.encoding.XMLType;
import org.apache.axis.transport.http.HTTPConstants;
import org.apache.axis.utils.Options;

import java.net.URL;
import java.util.*;
public class Axis2DotNet {
   public static void main(String args[]) {
      try {
         URL url =
          new URL("http://mindstrm.com:8199/CorpDataService/Proxy.asmx");
         Service  service = new Service();
         Call call = (Call) service.createCall();
         call.setTargetEndpointAddress( url );
```

8. Le nom réel est encore débatu, mais c'est pour l'instant Axis. Les exemples de ce livre sont basés sur la version Axis du 5 décembre 2001. Vous pouvez naturemment vous attendre à des changements au fur et à mesure du développement de l'API. Notez que la version alpha3 fonctionne : des exemples en français ont tous été compilés avec succès par le relecteur.

```
            call.setOperationName("getHeadLines");
            call.setProperty(Call.NAMESPACE,
                    "http://mindstrm.com/CorpDataService");
            call.setProperty(HTTPConstants.MC_HTTP_SOAPACTION,
                 "http://mindstrm.com/CorpDataService/getHeadLines");
            call.addParameter("symbole", XMLType.XSD_STRING, Call.PARAM_MODE_IN);
            call.setReturnType(XMLType.SOAP_ARRAY);
            ArrayList ret = (ArrayList)call.invoke(new Object[] {args[0]});
            Object[] res = ret.toArray();
            int cnt = res.length;
            for (int i = 0; i < cnt; i++) {
               System.out.println((String)res[i]);
            }
         }
         catch (Exception e) {
            System.out.println(e);
         }
      }
   }
```

Axis semble être une API plus structurée, ce qui représente un changement bienvenu par rapport à l'implémentation Apache SOAP. Ce n'est pas une attaque contre Apache SOAP. Ce dernier est un projet accompli et fonctionne très bien. L'API orientée objet qui émerge dans Axis n'aurait certainement jamais existé sans les leçons apprises lors du développement d'Apache SOAP.

Examinons-donc rapidement le code. Nous commençons par créer une instance de `java.net.URL`, comme nous le faisons dans les exemples Apache SOAP. L'étape suivante consiste à créer une instance de `org.apache.axis.client.Service`, qui est une sorte de fabrique d'objets `org.apache.axis.client.Call`, une instance créée en appelant `service.createCall()`. À partir de là, tout se fait *via* l'objet `Call`. Nous passons l'URL à `call.setTargetEndpointAddress()` et positionnons le nom de méthode en utilisant `call.setOperationName()`. L'étape suivante consiste à positionner l'espace de noms. Nous avons vu dans notre exemple précédent que l'espace de noms à utiliser était `http://mindstrm.com/CorpDataService`. L'espace de nom est une propriété de l'objet `Call` positionnée en utilisant la méthode `call.setProperty()`. La propriété espace de noms est spécifiée dans le premier paramètre en envoyant `Call.NAMESPACE` et la valeur est passée sous forme de chaîne dans le second. Nous avons également besoin de positionner la propriété `SOAPAction` de la même façon, cette fois en utilisant la constante `Call.MC_HTTP_SOAPACTION` comme nom de propriété et `http://mindstrm.com/CorpDataService/getHeadLines` comme valeur. Souvenez-vous, les services .NET exigent une valeur de `SOAPAction` correspondant à la méthode de service.

L'étape suivante consiste à positionner les paramètres du `Call`, en utilisant `call.addParameter()`. Le premier paramètre de cette méthode est le nom du paramètre à envoyer au service. Ce nom est utilisé comme nom d'élément dans l'enveloppe SOAP. Le second paramètre spécifie le type de données. Dans ce cas, la constante `XMLType.XSD_STRING` est utilisée pour indiquer que le paramètre est une instance du type chaîne standard. Le dernier paramètre, `Call.PARAM_MODE_IN`, indique que le paramètre est de type `in`. (Axis supporte également les paramètres `out` et `in/out`.) Il est intéressant de voir que les valeurs des paramètres ne sont pas spécifiées dans la méthode `call.addParameter()`. Au

lieu de cela, un `Object[]` est passé aux méthodes `call.invoke()` suivantes, dans lesquelles chaque entrée dans le tableau objet correspond à la valeur du paramètre déjà positionné à l'aide de `call.addParameter()`. Nous n'avons qu'un paramètre : la chaîne trouvée dans `args[0]`, qui contient le symbole de l'action. Nous utilisons donc `args[0]` pour créer une instance d'`Object[]` possédant une seule entrée.

La valeur de retour de la méthode `call.invoke()` est une instance de `java.util.ArrayList` contenant les nouvelles renvoyées pour le symbole d'action passé à la méthode de service. Appeler la méthode `toArray()` de l'`ArrayList` renvoie un `Object[]` dans lequel chaque objet est une `java.lang.String` contenant une nouvelle. Nous itérons dans ce tableau et affichons la nouvelle sur la console. J'ai exécuté `Axis2DotNet` en lui passant dans la ligne de commande. Voici le résultat que j'ai obtenu :

```
Autos: Year of Big Sales, Little Profits
Autos: Year of Zero Financing and Profits
Report: EchoStar May Invest in Spaceway
EchoStar may invest $1 billion in Spaceway-WSJ
GM-Daewoo Motor deal due by Jan 20 - creditor bank
In Detroit, a New Definition of 'Quality'
GM's bright idea; Ford's e-commerce plans take root
GM invests in Lutz's Cunningham Motors
December U.S. Auto Sales Seen Up Over '00'
RESEARCH ALERT-UBS Warburg ups GM EPS view
```

Il semble assez simple d'écrire des clients Axis pour des services .NET. Axis peut également utiliser à la volée des WSDL de services mais nous ne traiterons pas ce point.

Modifions le code pour accéder au service GLUE nommé `urn:CorpDataServices` exécuté sur *http://mindstrm.com:8004/glue*. Le nouveau nom de classe est `Axis2Glue`. Nous n'avons besoin que d'une modification dans la classe `Axis2DotNet` pour la faire fonctionner avec notre service GLUE : l'URL doit être modifiée en *http://tokyo:8004/glue/urn:CorpDataServices*. Tout le reste est inchangé. Tout cela est prometteur ! L'équipe Axis travaille apparemment à une normalisation de l'API et Axis peut se révéler être une implémentation très intéressante lorsqu'elle sera arrivée à maturité.

10

En-têtes SOAP

L'enveloppe SOAP peut contenir un élément Header qui englobe des données à l'extérieur des limites d'une invocation de style RPC (ou autre). L'en-tête est un mécanisme d'extension qui peut véhiculer n'importe quel type d'information située hors de la sémantique du message, mais elle est néanmoins utile voire nécessaire pour traiter correctement le message. L'information de routage peut contenir des identifiants de transaction, des clés d'authentification ou toute autre information apparentée au traitement du message ou à son routage.

Placer des informations de routage au sein de l'en-tête SOAP est utile lorsqu'un message est envoyé à un intermédiaire qui à son tour envoie le message vers sa destination finale. Au chapitre 9, nous avons développé des services proxy qui utilisaient un autre service pour effectuer leurs tâches. Dans des cas comme celui-ci, l'application cliente n'était pas informée du fait que le service auquel elle s'adressait était en réalité un proxy. Il est toutefois possible pour les services proxy de réacheminer des messages vers des destinations particulières ou de se servir d'autres destinations pour accomplir leurs tâches. Nous développerons un exemple de ce type de proxy ou service d'intermédiation.

Bien que SOAP ne spécifie pour l'instant aucun mécanisme de sécurité, l'élément Header est un bon endroit pour encoder des données d'authentification telles que des noms d'utilisateur et des mots de passe. Utiliser le Header pour stocker les données d'authentification vous permet de développer vos méthodes de service en utilisant uniquement les paramètres nécessaires et appropriés aux tâches de la méthode et vous fournit malgré tout un mécanisme pour envoyer les autres données au destinataire. Les sémantiques d'utilisation des en-têtes SOAP pour l'authentification sont dépendantes de l'implémentation, mais elles constituent un mécanisme raisonnable, en l'absence d'autre standard.

GLUE supporte particulièrement bien les en-têtes SOAP,[1] alors qu'Apache présente de vraies lacunes en la matière. Je vais vous guider à travers une technique d'utilisation des en-têtes dans les clients et services Apache SOAP.

1. La documentation GLUE est plus que suffisante pour vous permettre de traiter les en-têtes.

Fournisseurs et routeurs Apache SOAP

Le moteur Apache SOAP ne fournit pas de support direct pour les en-têtes SOAP. Cependant, il supporte suffisamment de classes Java pour vous permettre d'ajouter des en-têtes sans trop de difficultés. L'essentiel du travail se situera du côté du service. Fondamentalement, nous cherchons un moyen d'accéder à l'en-tête SOAP depuis la méthode de service que nous implémentons. Mais avant de nous plonger dans le code, voyons comment Apache SOAP traite et achemine les messages SOAP. La *figure 10-1* montre le chemin emprunté par un message RPC envoyé à un service Apache SOAP. Les objets grisés font partie de la structure Apache SOAP et ceux en blanc sont les classes Java développées dans ce livre.

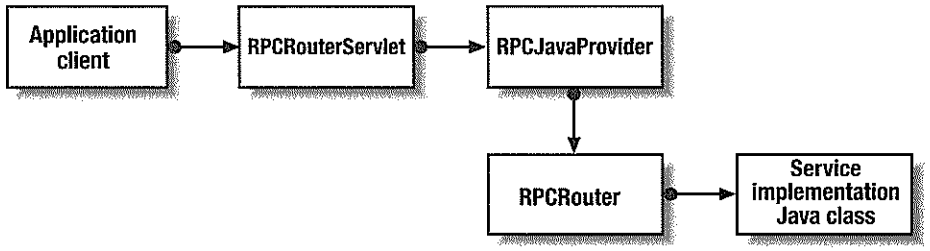


Figure 10-1. Chemin du client vers l'implémentation du service

Apache SOAP est basé sur des servlets Java. Lorsque vous envoyez un message SOAP de type RPC, vous envoyez en réalité le message à la classe `org.apache.soap.server.http.RPCRouterServlet`. Il appartient à la servlet de déterminer la classe appropriée à l'invocation. Tous nos services Apache SOAP écrits jusqu'à présent utilisaient le fournisseur Java par défaut. Voici la partie significative du descripteur de déploiement:

```
<isd:provider type="java">
```

Dans tous les cas, nous avons assigné à l'attribut `type` la valeur `"java"`. Ce n'est rien d'autre qu'un alias pour indiquer à la structure que nous voulons utiliser le fournisseur Java par défaut qui est `org.apache.soap.providers.RPCJavaProvider`. La servlet utilise cette classe fournisseur et toutes les classes fournisseurs pour localiser et invoquer la méthode de service. Le fournisseur par défaut utilise `org.apache.soap.server.RPCRouter` pour acheminer le message vers la classe d'implémentation du service appropriée. Le routeur charge la classe de service et invoque la méthode spécifiée à l'aide de la réflexion Java. Pour spécifier une autre implémentation de fournisseur, on assignera le nom de classe Java complet du fournisseur comme valeur de l'attribut `type`.

Remplacer des classes fournisseur et routeur

Pour mettre l'en-tête SOAP à disposition de nos méthodes de service, nous devrons remplacer la classe routeur par une classe de notre cru. Nous pourrions modifier la classe `org.apache.soap.server.RPCRouter` et reconstruire l'arborescence source Apache, mais cela serait un peu complexe. Nous devrions recommencer à chaque nouvelle version d'Apache. Fort heureusement, Apache SOAP utilise le concept de *fournisseurs connectables*. Cela signifie que vous pouvez connecter votre propre fournisseur de façon à

l'utiliser pour le déploiement d'un service donné. Pour utiliser votre propre connecteur, vous devez implémenter votre classe fournisseur. C'est exactement l'opportunité que nous cherchions. Créer notre fournisseur nous permet d'utiliser notre propre routeur qui pourra envoyer l'en-tête SOAP à notre méthode de service. Si nous implémentons par exemple une classe fournisseur nommée `FournisseurJavaRPCAmeliore`, nous pouvons l'associer à un déploiement de service en modifiant l'élément `isd:provider` dans le descripteur de déploiement approprié :

```
<isd:provider type="javasoap.livre.ch10.services.FournisseurJavaRPCAmeliore">
```

Cet élément indique à la servlet routeur RPC d'utiliser notre nouvelle classe fournisseur au lieu de la classe par défaut. C'est exactement ce que nous souhaitons. Écrivons donc notre fournisseur.

Tous les fournisseurs doivent implémenter l'interface `org.apache.soap.util.Provider`, qui contient les méthodes `locate()` et `invoke()`. La méthode `locate()` localise l'information de déploiement du service en préparation à l'invocation de la méthode. La méthode `invoke()` invoque la méthode de service, en supposant que l'appel préalable à `locate()` a été réussi.

Nous n'avons pas besoin ici de réinventer toute la roue. La classe `org.apache.soap.providers.RPCJavaProvider` fait exactement ce qu'il faut pour localiser le service et nous allons donc l'étendre afin de pouvoir modifier la méthode `invoke()` pour notre usage. Voici le code de notre nouveau fournisseur, `FournisseurJavaRPCAmeliore` :

```
package javasoap.livre.ch10.services;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.server.*;
import org.apache.soap.server.http.*;
import org.apache.soap.util.*;

public class FournisseurJavaRPCAmeliore
             extends org.apache.soap.providers.RPCJavaProvider {

   public void invoke(SOAPContext reqContext, SOAPContext resContext)
               throws SOAPException {
      try {
         Response resp = RouteurRPCAmeliore.invoke(envelope.getHeader(), dd,
               call, targetObject, reqContext, resContext);
         Envelope env = resp.buildEnvelope();
         StringWriter sw = new StringWriter();
         env.marshall(sw, call.getSOAPMappingRegistry(), resContext);
         resContext.setRootPart(sw.toString(),
                     Constants.HEADERVAL_CONTENT_TYPE_UTF8);
      }
      catch(Exception e) {
```

```
            if (e instanceof SOAPException)
               throw (SOAPException )e;

            throw new SOAPException(Constants.FAULT_CODE_SERVER, e.toString());
         }
      }
   }
```

J'ai dérivé le code de la méthode `invoke()` du code source de `org.apache.soap.providers.RPCJavaProvider`. Mes modifications apparaissent en gras. La première modification consiste à appeler la méthode statique `invoke()` de `RouteurRPCAmeliore`, qui reçoit une instance de `org.apache.soap.Header` en premier paramètre. Nous n'avons pas encore implémenté cette classe mais nous le ferons sous peu. Nous en avons besoin car le routeur par défaut, `org.apache.soap.server.RPCRouter`, ne nous permet pas d'envoyer des en-têtes à sa méthode `invoke()`. Les autres paramètres sont les mêmes que ceux de `org.apache.soap.server.RPCRouter`. Un des avantages d'étendre le fournisseur par défaut, est que celui-ci fait le nécessaire pour obtenir l'enveloppe SOAP dans son implémentation de la méthode `locate()`, qu'il stocke dans la variable de classe `envelope` (une instance de `org.apache.soap.Envelope`). Tout ce que nous avons à faire est donc d'envoyer `envelope.getHeader()` comme premier paramètre de la méthode `invoke()` de notre nouvelle classe routeur.

Nous pouvons maintenant implémenter notre routeur. Encore une fois, nous n'avons pas besoin d'écrire la classe entière. Au lieu de cela, nous allons étendre `org.apache.soap.server.RPCRouter`. Nous avons simplement besoin d'implémenter la méthode `invoke()` et nous pourrons nous satisfaire du reste de la classe de base. Notre méthode `invoke()` prendra un `org.apache.soap.Header` en premier paramètre. L'astuce sera de trouver un moyen de passer l'en-tête à nos méthodes de service.

L'un des intérêts d'avoir accès au code source, est que l'on peut parfois y découvrir des joyaux insoupçonnés. L'implémentation de `org.apache.soap.server.RPCRouter` attend en réalité que vos méthodes de service prennent une instance de `org.apache.soap.rpc.SOAPContext` comme premier paramètre. Dans ces circonstances, le routeur par défaut utilise simplement le reste des paramètres à partir de la position 1. Vous pouvez essayer cela si vous le souhaitez. Reprenez n'importe quelles classes de service Apache SOAP des précédents chapitres et insérez une instance de `org.apache.soap.rpc.SOAPContext` dans la signature de méthode en premier paramètre, poussant de ce fait le reste des paramètres vers la droite. Si vous reconstruisez et redéployez le service après cette modification, tout fonctionnera comme avant. Le routeur sait traiter la possibilité que ce paramètre spécial soit présent, mais il ne l'exige pas. Cette possibilité pourrait couvrir notre besoin d'accéder à l'en-tête SOAP. Mais obtenir l'en-tête depuis `org.apache.soap.rpc.SOAPContext` requiert beaucoup de travail. Pourtant, ce mécanisme est parfait pour nos besoins. Le résultat est un routeur qui nous permet d'implémenter des méthodes de service qui peuvent prendre une instance de `org.apache.soap.Header` en premier paramètre. Voici le code de la classe `RouteurRPCAmeliore`. Une fois encore, la plupart du code provient de la classe de base et les modifications apparaissent en gras.

```
package javasoap.livre.ch10.services;

import org.apache.soap.server.*;
```

```
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import org.w3c.dom.*;
import org.apache.soap.util.Bean;
import org.apache.soap.util.MethodUtils;
import org.apache.soap.util.IOUtils;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.util.StringUtils;

public class RouteurRPCAmeliore extends org.apache.soap.server.RPCRouter {

   public static Response invoke(Header hdr, DeploymentDescriptor dd,
                  Call call, Object targetObject, SOAPContext reqCtx,
                  SOAPContext resCtx)
         throws SOAPException {
      byte providerType = dd.getProviderType();
      Vector params = call.getParams();
      String respEncStyle = call.getEncodingStyleURI();
      Object[] args = null;
      Class[] argTypes = null;
      if (params != null) {
         int paramsCount = params.size();
         args = new Object[paramsCount];
         argTypes = new Class[paramsCount];
         for (int i = 0; i < paramsCount; i++) {
            Parameter param = (Parameter)params.elementAt(i);
            args[i] = param.getValue();
            argTypes[i] = param.getType();
            if (respEncStyle == null) {
               respEncStyle = param.getEncodingStyleURI();
            }
         }
      }

      if (respEncStyle == null) {
         respEncStyle = Constants.NS_URI_SOAP_ENC;


      }

      Bean result = null;
      try {
         if (providerType == DeploymentDescriptor.PROVIDER_JAVA ||
             providerType == DeploymentDescriptor.PROVIDER_USER_DEFINED) {
            Method m = null ;
            try {
               m = MethodUtils.getMethod (targetObject,
                     call.getMethodName(), argTypes);
```

```
        } catch(NoSuchMethodException e) {
           try {
              int paramsCount = 0 ;
              if (params != null) paramsCount = params.size();
              Class[] tmpArgTypes = new Class[paramsCount+1];
              Object[] tmpArgs = new Object[paramsCount+1];
              for (int i = 0 ; i < paramsCount ; i++)
                 tmpArgTypes[i+1] = argTypes[i] ;

              argTypes = tmpArgTypes;
              argTypes[0] = Header.class;
              m = MethodUtils.getMethod(targetObject,
                          call.getMethodName(),argTypes);
              for (int i = 0 ; i < paramsCount ; i++)
                 tmpArgs[i+1] = args[i];

              tmpArgs[0] = hdr;
              args = tmpArgs;
           }
           catch (NoSuchMethodException e2) {
              throw e;
           }
        catch (Exception e2) {
           throw e2;
        }
     }         catch (Exception e) {
        throw e;
     }

     result = new Bean(m.getReturnType(),
                       m.invoke(targetObject, args));
  }
  else {
     Class bc = Class.forName("org.apache.soap.server.InvokeBSF");
     Class[] sig = {DeploymentDescriptor.class,
                    Object.class,
                    String.class,
                    Object[].class};
     Method m = MethodUtils.getMethod(bc, "service", sig, true);
     result = (Bean) m.invoke (null,
               new Object[] {dd, targetObject,
                             call.getMethodName (), args});
  }
}
catch (InvocationTargetException e) {
   Throwable t = e.getTargetException();
   if (t instanceof SOAPException) {
      throw (SOAPException)t;
   }
   else {
```

```
            throw new SOAPException(Constants.FAULT_CODE_SERVER,
                "Exception from service object: " + t.getMessage(), t);
        }
      }
      catch (ClassNotFoundException e) {
         throw new SOAPException (Constants.FAULT_CODE_SERVER,
                 "Unable to load BSF: script services " +
                 "unsupported without BSF", e);
      }
      catch (Throwable t) {
         throw new SOAPException (Constants.FAULT_CODE_SERVER,
           "Exception while handling service request: " + t.getMessage(), t);
      }

      Parameter ret = null;
      if (result.type != void.class) {
         ret = new Parameter (RPCConstants.ELEM_RETURN, result.type,
                              result.value, null);
      }

      return new Response(call.getTargetObjectURI(), call.getMethodName (),
                              ret, null, null, respEncStyle, resCtx);
      }
   }
```

Le premier paramètre de la méthode `invoke()` est une instance de `org.apache.soap.Header` qui contient l'en-tête SOAP de l'enveloppe de la requête. À l'aide de l'introspection Java, une tentative est faite d'obtenir la méthode de la classe de service, en supposant que ses paramètres et leurs types associés correspondent directement à ceux trouvés dans le corps SOAP. En cas d'échec, on réessaie en recherchant cette fois une méthode possédant tous les paramètres du corps SOAP ainsi qu'un premier paramètre de type `org.apache.soap.Header`. Pour cela, nous assignons à `argTypes[0]` la valeur `Header.class` et à `tmpArgs[0]` la valeur `hdr` (le premier paramètre de notre méthode `invoke()`). Ces tableaux sont utilisés par le mécanisme de réflexion Java lors de la récupération et l'invocation dynamique de méthodes Java. Comme vous pouvez le voir, la majorité du code reste inchangée. L'essentiel a pour objet de retrouver une méthode de classe de service appropriée en tenant compte de la possibilité qu'un script BSF implémente le service,[2] et de traiter les fautes.

Service Apache SOAP prenant en charge les en-têtes SOAP

Nous avons maintenant un fournisseur et un routeur pouvant passer l'en-tête SOAP à nos méthodes de service et nous allons donc continuer et implémenter un service utilisant l'en-tête SOAP. Le `urn:ServiceProxyCotation` du chapitre 9 est un bon point de départ. Son implémentation code en dur l'adresse physique du service *http://minds-*

2. Apache SOAP vous permet d'implémenter des services à l'aide de scripts. Référez-vous à la documentation Apache SOAP sur ce sujet.

trm.com:8004/glue/urn:CorpDataServices. Nous allons demander à l'application cliente de fournir cette information dans l'en-tête SOAP. Notre service proxy renvoie vers la fausse adresse *http://mindstrm.com:8899/glue/urn:CorpDataServices* si aucune donnée de routage n'apparaît dans l'en-tête. Vous pouvez utiliser une technique similaire si le service accède à un service par défaut mais peut également être redirigé vers un service spécifique au client par l'intermédiaire de l'en-tête SOAP.

Concevons notre en-tête SOAP de façon à ce qu'elle contienne un seul élément enfant nommé `targetAddress`. Cet élément contient l'adresse du service de destination finale que le proxy doit utiliser. Les éléments de l'en-tête peuvent utiliser l'`attribut SOAP-ENV:mustUnderstand` avec la valeur `"1"` pour indiquer que le destinataire de l'enveloppe doit pouvoir interpréter l'élément en-tête et utiliser correctement les données fournies. L'en-tête SOAP envoyée par un client demandant l'interprétation de `targetAddress` doit ressembler à ceci :

```
<SOAP-ENV:Header>
<targetAddress SOAP-ENV:mustUnderstand="1">
http://mindstrm.com:8004/glue/urn:CorpDataServices
</targetAddress>
</SOAP-ENV:Header>
```

Concevons donc notre service pour qu'il sache interpréter l'élément d'en-tête `target-Address` et aucun autre. Si d'autres éléments de l'en-tête apparaissent avec pour valeur de l'attribut `mustUnderstand "1"`, le service renverra une faute SOAP. Le nouveau service s'appellera `urn:ServiceProxyCotationRedirige`, implémenté par la classe `Service-Proxy-CotationRedirige` :

```
package javasoap.livre.ch10.services;

import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;

import javasoap.livre.ch9.services.CotationDistante;
import javasoap.livre.ch9.services.ProxyCotation;

public class ServiceProxyCotationRedirige {

   static final String ADRESSECIBLE = "targetAddress";
   static final String CIBLEDEFAUT =
            "http://mindstrm.com:8899/glue/urn:CorpDataServices";

   public ServiceProxyCotationRedirige() {
   }

   String getTargetAddress(Header entete)
              throws SOAPException  {
```

```
        String cible = CIBLEDEFAUT;

        Vector entrees = entete.getHeaderEntries();
        for (int i = 0; i < entrees.size(); i++) {
           Element el = (Element)entrees.elementAt(i);
           String nom = el.getLocalName();
           String val = el.getAttributeNS(Constants.NS_URI_SOAP_ENV,
              Constants.ATTR_MUST_UNDERSTAND);

           if (nom.equals(ADRESSECIBLE)) {
              cible = el.getFirstChild().getNodeValue();
           }
           else if (val != null && val.equals("1")) {
              if (!nom.equals(ADRESSECIBLE))
              throw new SOAPException("MustUnderstand",
                "Le service ne peut interpréter l'élément en-tête : " + nom);
           }
        }

        return cible;
     }

     public ProxyCotation getCotation(Header hdr, String symbole)
                        throws SOAPException, Exception {

        String cible = getTargetAddress(hdr);

        URL url = new URL(cible);

        SOAPMappingRegistry smr = new SOAPMappingRegistry();

        BeanSerializer beanSer = new BeanSerializer();

        smr.mapTypes(Constants.NS_URI_SOAP_ENC,
          new QName(
           "http://www.themindelectric.com/package/javasoap.livre.ch9.services/",
           "Quote"),
           CotationDistante.class, beanSer, beanSer);



        Call appel = new Call();
        appel.setSOAPMappingRegistry(smr);
        appel.setTargetObjectURI("XYZ");
        appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        String action = symbole;

        Vector params = new Vector();
```

```
        params.addElement(new Parameter("action", String.class, action, null));
        appel.setParams(params);

        appel.setMethodName("getCotation");
        Response rep = appel.invoke(url, "");
        ProxyCotation cotation = new ProxyCotation();
        if (rep.generatedFault()) {
           throw new Exception("Echec dans l'appel au service");
        }
        else {
           Parameter ret = rep.getReturnValue();
           CotationDistante valeur = (CotationDistante).getValue();
           cotation.setSymboleAction(valeur.get_symbole());
           cotation.setDernier(valeur.get_dernierPrix());
           cotation.setDiff(valeur.get_change());
           cotation.setDate(valeur.get_dernieremodif());
           cotation.setVol(valeur.get_volume());
        }

        return cotation;
     }
  }
```

Le code est globalement le même que celui de la classe `ProxyCotation` développée au chapitre 9. La méthode `getCotation()` reçoit maintenant en premier paramètre un `org.apache.soap.Header` et une `String` contenant le symbole de l'action en second paramètre. Avant de construire l'URL, on effectue un appel à la méthode `getTargetAddress()`, l'en-tête étant passé en paramètre. La méthode `getTargetAddress()` renvoie soit la fausse adresse par défaut si aucun élément `targetAddress` n'apparaît dans l'en-tête, soit la valeur de l'élément `targetAddress` s'il existe. Nous itérons dans tous les éléments de l'en-tête à la recherche de l'élément `targetAddress`. Pendant que nous y sommes, nous vérifions si l'un des autres éléments contient un attribut `SOAP-ENV:mustUnderstand` possédant la valeur `"1"` et renvoyons une `SOAPException` appropriée si c'est le cas. L'envoi de l'exception se traduit par une faute SOAP correspondante envoyée à l'appelant. La valeur de `getTargetAddress()` renvoyée est utilisée pour construire l'URL. Le reste du code est le même que celui de `javasoap.livre.ch9.services.ServiceProxyCotation`.

Déployons le service en pensant à utiliser notre classe `FournisseurJavaRPCAmeliore` comme valeur de l'attribut type de l'élément provider. Voici à quoi le descripteur de déploiement doit ressembler :

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:ServiceProxyCotationRedirige">
  <isd:provider type="javasoap.livre.ch10.services.FournisseurJavaRPCAmeliore"
                scope="Application"
                methods="getCotation">
  <isd:java class="javasoap.livre.ch10.services.ServiceProxyCotationRedirige"
        static="false"/>
  </isd:provider>
```

```
    <isd:faultListener>org.apache.soap.server.DOMFaultListener
    </isd:faultListener>

    <isd:mappings>
      <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           xmlns:x="urn:ServiceProxyCotation" qname="x:Cotation"
           javaType="javasoap.livre.ch9.services.ProxyCotation"
           java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
           xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
    </isd:mappings>
  </isd:service>
```

Écrivons maintenant une application cliente Apache SOAP accédant à `urn:Service-ProxyCotationRedirige`. L'essentiel du code nous est maintenant familier et je décrirai donc seulement les parties nouvelles relatives à l'en-tête SOAP. La classe se nomme `ClientEntete`:

```
package javasoap.livre.ch10.clients;

import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;

import javasoap.livre.ch9.services.ProxyCotation;

public class ClientEntete {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL("http://localhost:8080/soap/servlet/rpcrouter");

      SOAPMappingRegistry smr = new SOAPMappingRegistry();

      BeanSerializer beanSer = new BeanSerializer();

      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
        new QName("urn:ServiceProxyCotation", "Cotation"),
                ProxyCotation.class, beanSer, beanSer);

      Call appel = new Call();
      appel.setSOAPMappingRegistry(smr);
      appel.setTargetObjectURI("urn:ServiceProxyCotationRedirige");
      appel.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

      String actions = args[0];
```

```
Header entete = new Header();
String must = Constants.NS_PRE_SOAP_ENV + ":" +
                        Constants.ATTR_MUST_UNDERSTAND;

Vector entrees = new Vector();
DocumentBuilder constructeur = XMLParserUtils.getXMLDocBuilder();
Document doc = constructeur.newDocument();
Element elem = doc.createElement("targetAddress");
elem.setAttribute(must, "1");
Text txt = doc.createTextNode("");
txt.setData("http://mindstrm.com:8004/glue/urn:CorpDataServices");
elem.appendChild(txt);
entrees.addElement(elem);
entete.setHeaderEntries(entrees);
appel.setHeader(entete);

Vector params = new Vector();
params.addElement(new Parameter("actions", String.class, actions, null));
appel.setParams(params);

try {
   appel.setMethodName("getCotation");
   Response rep = appel.invoke(url, "");

   if (rep.generatedFault()) {
      Fault faute = rep.getFault();
      String code = faute.getFaultCode();
      String desc = faute.getFaultString();
      System.out.println(code + ": " + desc);

      Vector v = faute.getDetailEntries();
      int cnt = v.size();
      for (int i = 0; i < cnt; i++) {
         Element n = (Element)v.elementAt(i);
         Node nd = n.getFirstChild();
               + ": " + nd.getNodeValue());
      }
   }
   else {
      Parameter ret = rep.getReturnValue();
      ProxyCotation valeur = (ProxyCotation)ret.getValue();
      System.out.println("Symbole :     " + valeur.SymboleAction() +
                "\nDernier cours : " + valeur.getDernier() +
                "\nChange :     " + valeur.getDiff() +
                "\nDerniere modif : " + valeur.getDate() +
                "\nVolume :     " + valeur.getVol());
   }
}
catch (SOAPException e) {
```

```
            System.err.println("SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

Nous créons une instance de `org.apache.soap.Header` appelée `entete` que nous utiliserons pour construire les éléments d'en-tête. La variable `String` `doit` contenir le nom qualifié de l'attribut `mustUnderstand`. Nous construisons le nom à l'aide des valeurs des constantes `Constants.NS_PRE_SOAP_ENV` et `Constants.ATTR_MUST_UNDERSTAND`, séparées par un « : ». Ceci nous donne la valeur de `SOAP-ENV:mustUnderstand`. Nous avons maintenant à travailler au niveau du XML, comme nous l'avons fait lorsque nous avons développé nos entrées détaillées de fautes personnalisées au chapitre 7. Nous adopterons ici la même approche, hormis le fait que nous n'avons pas besoin de construire l'élément `SOAP-ENV:Header`. Ceci est laissé aux bons soins de la classe `org.apache.soap.Header`. Nous avons juste besoin de construire les entrées enfants dans l'en-tête et nous indiquons pour cela `targetAddress` comme premier élément du document. Nous positionnons ensuite l'attribut `mustUnderstand` en appelant `elem.setAttribute()` en lui passant la variable chaîne de caractère `must` avec la valeur `"1"`. Nous créons ensuite un nœud texte contenant comme données `http://mindstrm.com:8004/glue/urn:CorpDataServices`, qui est la destination finale que nous demandons au service proxy d'utiliser. Les entrées d'en-tête de type vecteur sont ajoutées à l'en-tête en appelant `header.setHeaderEntries()` et l'en-tête est ensuite ajoutée à l'objet appel en utilisant `appel.setHeader()`. L'en-tête que nous avons construite va maintenant être incluse lorsque l'enveloppe SOAP sera créée. J'ai exécuté cette application cliente en ligne de commande avec `AMX` comme valeur, produisant comme effet l'envoi de cette enveloppe SOAP au service :

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Header>
<targetAddress SOAPENV:mustUnderstand="1">
http://tokyo:8004/glue/urn:CorpDataServices
</targetAddress>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<ns1:getCotation
xmlns:ns1="urn:ServiceProxyCotationRedirige"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<actions xsi:type="xsd:string">AMX</actions>
</ns1:getCotation>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

11

JAX-RPC et JAXM

Dans la technologie des logiciels, la standardisation survient une fois un certain élan acquis. SOAP et les services web ont maintenant l'élan nécessaire et les standards (ainsi que leurs acronymes) commencent à fleurir. De nombreuses API destinées à la manipulation de XML en Java se développent et toutes se trouvent regroupées sous les Java *API for XML* (JAX). JAX n'est pas un produit, c'est une définition d'API et en cela n'est pas différent des innombrables spécifications d'API associées à Java. Toutefois, XML couvrant tant de domaines, JAX est fait de nombreux composants. Deux d'entre eux sont particulièrement intéressants pour nous: JAX-RPC et JAXM. JAX-RPC est l'API Java pour le RPC basé sur XML. C'est cette API que j'espère voir respecter les plus significatives des implémentations SOAP RPC. JAXM est l'API Java pour les messages XML et, comme JAX-RPC, on peut s'attendre à ce qu'elle soit largement acceptée.

Ces deux API sont nouvelles et ne sont pas encore prêtes pour le monde de la production. Les spécifications pour JAXM et JAX-RPC sont en cours de relecture publique et sont donc encore sujettes à modifications. À ce stade toutefois, les changements devraient être mineurs. Sun a cependant produit une implémentation de référence qui vous permet de commencer à travailler avec ces technologies. Cette implémentation est incluse dans le *Java Web Services Developer Pack* (Java WSDP) téléchargeable à *http://java.sun.com/webservices/webservicespack.html*. Vous pouvez considérer cette version comme une première bêta. Tant que vous ne l'utilisez pas dans un cadre de production et que vous ne vous attendez pas à voir tous les bogues corrigés, elle vous aidera à gagner du temps dans la perspective de monter un environnement de production une fois finalisée. Le Web Services Developer Pack utilise un serveur Tomcat 4; si vous avez utilisé ce dernier avec Apache SOAP jusqu'à présent, vous devriez vous sentir à l'aise avec JAX. Même si vous n'avez pas utilisé Tomcat, les instructions d'installation du pack sont simples et vous ne devriez pas rencontrer de difficultés.

Pour explorer JAX-RPC, nous allons construire quelques exemples en utilisant l'implémentation de référence. Ces exemples seront de nouvelles versions d'exemples développés précédemment dans ce livre. Nous avancerons rapidement, en supposant que vous avez lu les chapitres précédents. Nous n'essaierons pas de couvrir JAX-RPC et JAXM de façon exhaustive. Au lieu de cela, nous utiliserons cette opportunité pour jeter un œil rapide sur la façon dont ces API sont structurées et comment elles se comparent aux API

Apache SOAP 2.X et GLUE vues jusqu'à maintenant. En d'autres termes, nous allons reconstruire avec JAX quelques uns des exemples de services et de clients des chapitres précédents.

JAX-RPC

Commençons avec JAX-RPC, puisque nous avons passé le plus clair de notre temps dans le monde RPC. Tout comme GLUE, JAX-RPC masque les détails du protocole sous-jacent (SOAP) au programmeur. Pour cela, il utilise des objets désignés sous le nom de stubs et de ties. Un *stub* est utilisé par une application cliente pour accéder à un service distant. Le stub ressemble à une interface de service, mais il s'exécute hors du processus client local. Il utilise la structure sous-jacente pour générer un message SOAP approprié et l'envoyer par HTTP. Ainsi, vu depuis l'angle du programmeur d'applications clientes, si vous avez un stub, vous n'avez pas à vous préoccuper des détails du message SOAP. La *figure 11-1* montre les relations entre le code client, les stubs et les composants d'exécution de la structure cliente JAX-RPC.

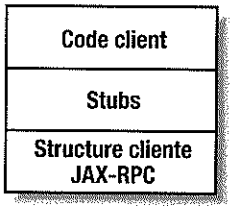


Figure 11-1. Les stubs dans l'architecture JAX-RPC

Un *tie* est conceptuellement similaire à un stub, mais se trouve côté serveur. Il agit comme un pont entre le moteur d'exécution JAX-RPC et l'implémentation du service et découple le code de votre service de la structure JAX-RPC. Les messages de services actifs sont pris en charge par le moteur d'exécution et passés au service *via* un objet *tie*. Les réponses des services repartent par le tie vers le moteur JAX-RPC, où ils sont convertis en réponse SOAP ou en messages de fautes SOAP puis renvoyés par HTTP. La *figure 11-2* montre les relations entre le code du service, les ties et le moteur JAX-RPC.

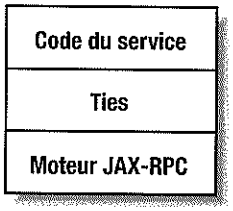


Figure 11-2. Les ties dans l'architecture JAX-RPC

Réfléchissons un moment aux mécanismes RPC dans Apache SOAP 2.X et GLUE. Apache expose une grande part de la machinerie SOAP au programmeur Java et demande à celui-ci de traiter SOAP à un niveau concret. Du point de vue de l'architecture, JAX-RPC ressemble un peu plus à GLUE dans le sens où il masque la plupart des détails de bas niveau. Ceci est une tendance positive dans l'évolution des API SOAP qu'elles soient ou non propriétaires.

Quelle que soit la complexité de l'API ou du type de mécanisme distribué que vous choisissez, il est souhaitable d'abstraire et d'isoler les parties de votre code qui prennent en charge ces mécanismes. Cela vous permet de passer d'un mécanisme à l'autre avec peu d'impact sur le reste de votre travail. Les exemples de ce chapitre utilisent cette approche même s'ils sont tellement simples que cela semble excessif. La *figure 11-3* montre l'architecture d'ensemble des exemples RPC que nous allons développer avec JAX-RPC.

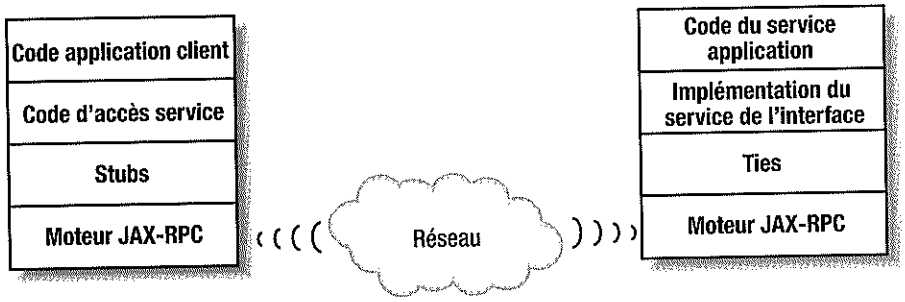


Figure 11-3. Architecture RPC utilisant JAX-RPC

Travailler sans Ant

Le didacticiel du *Java Web Services Developer Pack* s'appuie beaucoup sur un package nommé Ant. Toute la compilation des fichiers source Java, la création des répertoires, la création des fichiers JAR, etc., sont prises en charge par les fichiers de configuration de Ant. Bien que Ant soit assez répandu et que son utilisation pour la fabrication de code génération de projets tel que celui de ce chapitre soit plutôt une bonne idée, nous ne traiterons pas de son usage ici. (Pour plus d'information sur Ant, voir *Ant: The Definitive Guide* de Jesse Tilly et Eric Burke.) Au lieu de cela, je vous guiderai dans les étapes manuelles au travers de ces exemples. Si vous savez utiliser Ant, vous serez capable de passer sur certains détails de ce chapitre.

Il faut commencer par renseigner le CLASSPATH. La pré-version utilise 10 fichiers JAR qui doivent y figurer. Ces fichiers sont situés sous le répertoire *common\lib* à la racine de votre installation du Developer Pack et il vous appartient de vous assurer qu'ils sont dans votre classpath. Ces fichiers sont :

jaxrpc-ri.jar
jaxrpc-api.jar
activation.jar
dom4j.jar
jaxm-api.jar
jaxm-client.jar
log4j.jar
mail.jar
xalan.jar
xerces.jar

Créer un service JAX-RPC

Au chapitre 9, nous avons parlé de la construction de services proxy situés entre l'application cliente et d'autres services. Dans ce chapitre, le service final avec lequel les proxies communiquent était situé à l'adresse *http://mindstrm.com:8004/glue/urn:CorpDataServices*. Nous allons faire de même mais nous allons cette fois bâtir notre service proxy avec l'implémentation de référence JAX-RPC.[1] Cet exemple nous permet de voir comment JAX-RPC et GLUE interopèrent et nous donne l'opportunité de batir à la fois un service et un client avec JAX-RPC.

La première étape pour la construction du service consiste à écrire l'interface Java que j'appellerai `javasoap.livre.ch11.services.IServiceProxyAction`. Cette interface doit implémenter l'interface `java.rmi.Remote` car JAX-RPC utilise le paquetage Java RMI. Cette stratégie est assez courante car Java RMI fournit une bonne abstraction de l'informatique distribuée. Les méthodes de l'interface interface doivent être définies de façon à renvoyer une `RemoteException`, qui vient également avec le paquetage `java.rmi`. Nous définirons une méthode, `getCotationAction()`, recevant un paramètre chaîne pour le symbole de l'action et renvoyant une instance de `javasoap.livre.ch9.services.ProxyCotation`. Il n'est pas nécessaire de créer une nouvelle classe pour renvoyer la cotation. Nous l'avons déjà fait au chapitre 9. Voici à quoi le code de l'interface `IServiceProxyAction` ressemble :

```
package javasoap.livre.ch11.services;

import java.rmi.Remote;
import java.rmi.RemoteException;
import javasoap.livre.ch9.services.ProxyCotation;

public interface IServiceProxyAction extends Remote {
    public ProxyCotation getCotationAction(String symbole) throws RemoteException;
}
```

Nous devons ensuite écrire une classe Java qui implémente l'interface `IServiceProxyAction`, que j'appelerai `javasoap.livre.ch11.services.ServiceProxyActionImpl`. Je n'utilise pas normalement le suffixe `Impl` dans les noms de mes classes, mais c'est une pratique courante et convient à mes besoins ici. (Habituellement, je supprime simplement le « I » du nom de l'interface et appelle la classe d'implémentation `ServiceProxyAction`, mais je préfère utiliser cette forme pour autre chose. Je veux découpler la classe qui effectue vraiment le travail des classes qui utilisent JAX-RPC ou toute autre API.) Ainsi, la classe `ServiceProxyActionImpl` fait partie de la couche d'implémentation de l'interface de service de la *figure 11-3*. Nous utiliserons une autre classe, nommée `javasoap.livre.ch11.services.ServiceProxyAction`, pour faire le travail réel. À la différence de `ServiceProxyActionImpl`, cette classe ignore le mécanisme JAX-RPC ainsi que l'interface `IServiceProxyAction` utilisée par l'objet tie. Cela nous permet d'utiliser la classe sans modification, indépendamment du mécanisme distribué que nous utilisons. En d'autres termes, `ServiceProxyAction` fait partie de la couche code de l'application de la *figure 11-3*. Nous utiliserons une instance de `ServiceProxyAction` dans la

1. Je suppose que vous avez téléchargé et installé le Java Web Services Developer et vérifié qu'il fonctionnait correctement.

méthode `getCotationAction()` de la classe `ServiceProxyActionImpl` comme on le voit ci-dessous :

```
package javasoap.livre.ch11.services;

import javasoap.livre.ch9.services.ProxyCotation;

public class ServiceProxyActionImpl implements IServiceProxyAction {
ServiceProxyAction sp;[NDR]

    public ProxyCotation getCotationAction(String symbole) {
      if(sp==null) sp = new ServiceProxyAction();
        return sp.getCotationAction(symbole);
    }
}
```

Nous allons maintenant implémenter l'interface `ServiceProxyAction`, dont le travail consiste à invoquer la méthode `GetCotation()` sur le service `urn:CorpDataServices` service. Toutefois, je veux différer la communication avec ce service GLUE car cela implique l'utilisation de son fichier WSDL. Nous y reviendrons plus tard. Pour l'instant, la méthode `getCotationAction()` de `ServiceProxyAction` renverra juste une instance de `ProxyCotation` avec des valeurs factices. Voici le code :

```
package javasoap.livre.ch11.services;

import javasoap.livre.ch9.services.ProxyCotation;

public class ServiceProxyAction {

        public ProxyCotation getCotationAction(String symbole) {

        ProxyCotation pq = new ProxyCotation();
        pq.setStockSymbol(symbole);

        // utilisation de valeurs factices pour l'instant
        pq.setDernier((float)0.0);
        pq.setDiff((float)0.0);
        pq.setDate("NA");
        pq.setVol(0);

        return pq;

      }
  }
```

Ce sont les seules classes et interfaces Java dont nous avons besoin pour implémenter le service. Continuez et compilez-les. Il est maintenant temps de générer les stubs et les ties pour le service. Un outil en ligne de commande nommé `xrpcc` génère ces classes. `xrpcc` utilise en entrée un fichier de configuration. Ce fichier est un fichier XML contenant les informations sur les classes, les espaces de noms, les noms de services etc. Je l'ai nommé *CorpDataServices.xml*. Examinons son contenu :

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
  <rmi name="CorpDataServicesProxy"
    targetNamespace="http://mindstrm.com/wsdl"
    typeNamespace="http://mindstrm.com/types">
      <service name="CorpDataServices"
          packageName="javasoap.livre.ch11.client">
        <interface name="javasoap.livre.ch11.services.IServiceProxyAction"
         servantName="javasoap.livre.ch11.services.ServiceProxyActionImpl"/>
      </service>
  </rmi>
</configuration>
```

Tout se trouve à l'intérieur de la balise `configuration`, qui utilise l'attribut `xmlns` pour définir l'espace de noms global comme étant `http://java.sun.com/jax-rpc-ri/xrpcc-config`. Dans la mesure où nous utilisons une interface basée sur RMI, le premier enfant est nommé `rmi`.[2] L'attribut `name` est utilisé pour spécifier le nom du *modèle*, qui est essentiellement le nom de base qui sera utilisé pour les fichiers générés par ce processus. Vous pouvez le considérer comme le nom de la définition du service. (Il n'est utilisé que pour cela dans le fichier WSDL qui est généré.) Les attributs `targetNamespace` et `typeNamespace` sont utilisés pour déclarer les espaces de noms qui seront associés respectivement au service cible et aux types personnalisés. Sous la balise `rmi` se trouve la balise `service`, qui fournit à `xrpcc` l'information sur les classes Java et les interfaces utilisées pour implémenter le service. L'attribut `name` contient le nom du service lui-même et l'attribut `packageName` contient le nom du paquetage Java à utiliser pour générer le code côté client. Sous `service` se trouve la balise `interface`, qui indique à l'outil quelles classes implémentent l'interface du service. L'attribut `name` contient le nom pleinement qualifié de l'interface Java du service et le `servantName` contient le nom pleinement qualifié de la classe Java qui implémente cette interface.

Continuons et exécutons `xrpcc` pour notre service. La racine de mon classpath de travail est `e:\projects\soap` et je l'indique dans la commande comme ceci :

```
xrpcc -both -keep -classpath e:\projects\soap -d gen CorpDataServices.xml
```

L'option `-both` indique à `xrpcc` que nous voulons qu'il génère les fichiers à la fois côtés client et serveur. Si pour une raison quelconque vous souhaitez les générer séparément, vous devrez utiliser à la place `-client` ou `-server`. Si vous spécifiez `-keep`, comme je l'ai fait, les fichiers source Java du code généré ne seront pas supprimés. (Par défaut, `xrpcc` compile le code généré puis supprime les fichiers source dans la mesure où vous n'en avez pas vraiment besoin. J'ai utilisé `-keep` de façon à pouvoir jeter un coup d'œil au code. Il est utile de voir quelles classes et méthodes `xrpcc` génère, particulièrement lorsque vous débutez.) L'option `-classpath` spécifie le classpath à utiliser pour trouver les fichiers d'entrée des classes. L'option `-d` indique quel répertoire racine utiliser pour les fichiers générés. Enfin, le dernier argument de la ligne est le nom du fichier de configuration.

---

2. `xrpcc` permet également l'utilisation de documents WSDL au lieu de RMI. Nous utiliserons cette approche plus loin dans ce chapitre.

Voici une liste des fichiers générés par `xrpcc`, organisés par sous-répertoires dans le répertoire de travail de `xrpcc`. Les éléments en gras sont des noms de répertoires. Pour plus de clarté, je n'ai pas listé les fichiers source Java conservés par l'option `-keep`. (En fait, je les ai détruits manuellement pour m'assurer qu'ils ne seraient pas inclus lors du déploiement.)

gen
- CorpDataServices.wsdl
- CorpDataServicesProxy_Config.properties
- **javasoap\livre\ch11\client**
  - CorpDataServicesProxy.class
  - CorpDataServicesProxy_SerializerRegistry.class
  - CorpDataServicesProxyImpl.class
- **javasoap\livre\ch11\services**
  - GetCotationAction_RequestStruct.class
  - GetCotationAction_RequestStruct_SOAPSerializer.class
  - GetCotationAction_ResponseStruct.class
  - GetCotationAction_ResponseStruct_SOAPBuilder.class
  - GetCotationAction_ResponseStruct_SOAPSerializer.class
  - IServiceProxyAction_Stub.class
  - IServiceProxyAction_Tie.class
- **javasoap\livre\ch9\services**
  - ProxyCotation_SOAPSerializer.class

J'ai placé les fichiers dans un répertoire séparé *gen* car ils seront archivés dans un fichier JAR un peu plus tard et que je ne souhaite pas archiver tout mon répertoire `javasoap.livre`. La première fois que j'ai utilisé `xrpcc`, j'ai remarqué que les classes stub et tie avaient été générées dans le même paquetage. Je n'aime pas procéder de la sorte. Comme vous avez pu le constater, j'ai maintenu mes classes client et serveur dans des paquetages séparés mais `xrpcc` n'offre pas cette option. De plus, la classe sérialiseur `ProxyCotation_SOAPSerializer` pour la classe `javasoap.livre.ch9.services.ProxyCotation` a été placée dans le paquetage `javasoap.livre.ch9.services`. Je suppose que cela à un sens bien que ce n'est pas là que je voudrais placer le sérialiseur. Mais là encore, je n'ai aucun contrôle. On pourrait certainement s'amuser avec les noms de paquetages dans le code source que `xrpcc` génère et tout recompiler à la main, mais je ne vous ennuierai pas avec ça. Encore une fois, il s'agit d'une pré-version et les choses évoluent. Souvenez-vous simplement que la structure des paquetages créée par `xrpcc` peut ne pas vous satisfaire.

Nous devons ensuite créer un descripteur de déploiement. L'information qu'il contiendra sera utilisée pour configurer l'application sur le serveur web. Voici le descripteur pour notre service, qui doit être nommé *web.xml*. Je le place dans le répertoire *gen*. Le texte en gras représente le code spécifique à cet exemple.

```
<?xml version="1.0" encoding="UTF-8"?>

 <!DOCTYPE web-app PUBLIC
     "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
     "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
   <display-name>CorpDataServices</display-name>
   <description>Corporate Data Services</description>
   <servlet>
      <servlet-name>JAXRPCEndpoint</servlet-name>
      <display-name>JAXRPCEndpoint</display-name>
      <description>
         Destination finale des services de données d'entreprise
      </description>
      <servlet-class>
        com.sun.xml.rpc.server.http.JAXRPCServlet
      </servlet-class>
      <init-param>
         <param-name>configuration.file</param-name>
         <param-value>
          /WEB-INF/CorpDataServicesProxy_Config.properties
        </param-value>
      </init-param>
      <load-on-startup>0</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>JAXRPCEndpoint</servlet-name>
      <url-pattern>/jaxrpc/*</url-pattern>
   </servlet-mapping>
   <session-config>
      <session-timeout>60</session-timeout>
   </session-config>
</web-app>
```

Nous n'avons pas encore terminé. Maintenant que nous avons implémenté les classes et les interfaces Java pour notre service, généré les ties et les stubs et créé un descripteur de déploiement, l'étape suivante consiste à archiver le service dans un fichier WAR. Un fichier WAR est une archive d'application web, c'est-à-dire un fichier JAR contenant les fichiers nécessaires au déploiement et à l'exécution d'une application web selon les spécifications de la servlet Java. Nous allons copier les fichiers de classes pour `IServiceProxyAction`, `ServiceProxyActionImpl` et `ServiceProxyAction` dans un répertoire approprié sous le répertoire *gen* et faire de même pour la classe `javasoap.livre.ch9.services.ProxyCotation`. Nous avons besoin d'organiser nos fichiers et nos répertoires pour fabriquer un fichier WAR correct. Tout doit partir d'un répertoire racine qui doit être nommé *WEB-INF* (le tout en majuscules). Déplacez les fichiers *web.xml*, *CorpDataServicesProxy_Config.properties* et *CorpDataServices.wsdl* dans le répertoire *WEB-INF*. Nous devons maintenant placer les classes Java dans les bons répertoires. Créez un répertoire nommé *classes* sous *WEB-INF* et faites-en la racine du répertoire *javasoap*. Autrement dit, déplacez le répertoire *javasoap* et tout son contenu sous *WEB-*

INF\classes. Ant rendrait le processus un peu plus simple : vous n'auriez qu'à paramétrer quelques fichiers de configuration et il s'occuperait du reste. Mais maintenant, vous savez ce que Ant effectue en coulisses.

Créons notre fichier WAR. À partir du répertoire *gen*, entrez la commande :

```
jar cvf corpdata.war WEB-INF
```

Cela archive tout ce dont nous avons besoin dans un fichier nommé *corpdata.war*. La *figure 11-4* montre le contenu du fichier WAR à l'aide du programme WinZip.

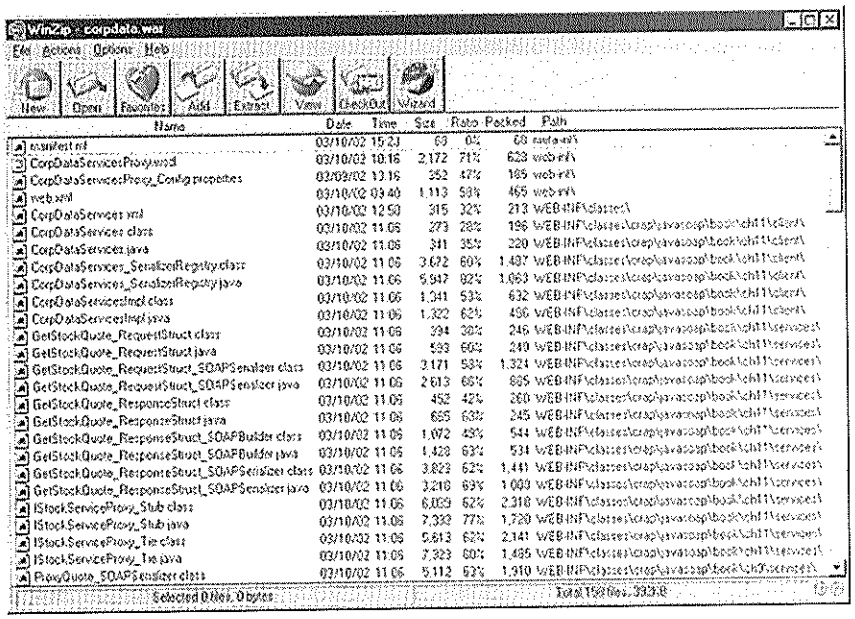


Figure 11-4. Contenu de corpdata.war

En avons-nous terminé ? Pas tout à fait. La dernière étape consiste à déployer le service, ce qui revient à copier le fichier WAR vers le répertoire *webapps* de Tomcat 4 (souvenez-vous que Tomcat 4 est inclus dans le Java Web Services Developer Pack). Faites-le. Mon Web Services Developer Pack est installé sous *e:\jwsdp-1_0-ea1*, j'ai donc copié *corpdata.war* sous *e:\jwsdp-1_0-ea1\webapps*. Si Tomcat 4 tournait, redémarrez-le. Cela lui fera lire le fichier *corpdata.war* et déployer le service. Vous pouvez vous assurer que le service est correctement déployé en accédant à *http://localhost:8080/corpdata/jaxrpc* dans votre navigateur. Si le service est bien installé, vous verrez ceci :

```
A Web Service is installed at this URL.
It supports the following ports: "IServiceProxyAction"
   (http://localhost:8080/corpdata/jaxrpc/IServiceProxyAction)
```

Parfait, cela a été bien plus facile que l'on aurait pu le croire. L'essentiel du travail consistait à archiver et à déployer. Si vous aviez utilisé Ant, cela aurait été encore plus vite. Souvenez-vous qu'il s'agit de la pré-version d'une implémentation de référence et vous pouvez être certain que les systèmes de production contiendront de meilleurs outils de déploiement.

Maintenant que notre service est déployé, réfléchissons un moment. Le processus de déploiement est-il meilleur que celui utilisé sous Apache SOAP. Il semblerait. Nous n'avons pas eu à écrire de code autre que celui qui a servi à implémenter le service alors que sous Apache SOAP, nous avions à en écrire une partie relative au transport, à la mise en correspondance des types etc. En particulier, remarquez que nous n'avons pas eu à nous occuper directement de notre classe personnalisée `ProxyCotation`. Les stubs et les ties se sont entièrement chargés de la mise en correspondance et du processus de sérialisation.

Comment comparer cela à GLUE en termes de déploiement de service ? C'est assez similaire bien qu'il y ait quelques subtiles mais frappantes différences. En tant que développeur, je sais que j'aurais certainement préféré ne pas avoir à m'occuper d'un quelconque code généré. Pourquoi ai-je à gérer ces fichiers de classes ? Sous GLUE, une partie de cela est pris en charge dynamiquement, ce qui rend la gestion un peu plus simple. Vous aurez toujours besoin de quelques stubs côté client mais le développement du service masquera tout. À mon avis, c'est là un avantage.

Créer un client JAX-RPC

Avant de commencer à écrire une application cliente, assurons-nous d'avoir copié dans les bons répertoires tous les fichiers de classes générés précédemment. Souvenez-vous qu'ils se trouvent dans un fichier WAR et sous le répertoire *gen\WEB-INF\classes*. Mais aucun de ces deux endroits ne se trouve dans mon classpath local et le moment est venu de les placer là où il faut pour que l'application cliente puisse y accéder.

Écrire une application cliente pour ce service est assez simple. Tout ce que vous avez à faire est d'obtenir une instance du stub qui a été généré plus tôt et d'appeler la méthode de service souhaitée sur cet objet. Voici le code d'un exemple de classe nommée `javasoap.livre.ch11.client.AppCotationAction` :

```
package javasoap.livre.ch11.client;

import javasoap.livre.ch11.services.*;
import javasoap.livre.ch9.services.*;

public class AppCotationAction {
   public static void main(String[] args) throws Exception {
      CorpDataServicesProxyImpl proxy =
         new CorpDataServicesProxyImpl();
      IServiceProxyAction_Stub stub =
         (IServiceProxyAction_Stub)proxy.getIServiceProxyAction();
      stub._setTargetEndpoint(args[0]);
      ProxyCotation cotation = stub.getCotationAction("IBM");
      System.out.println("Prix : " + cotation.getDernier());
      System.out.println("Volume : " + cotation.getVol());
      System.out.println("Dernière modif : " + cotation.getDate());
      System.out.println("Différence : " + cotation.getDiff());
   }
}
```

Nous créons une instance de `CorpDataServicesProxyImpl`, puis nous appelons sa méthode `getIServiceProxyAction()`. Ceci nous donne une référence au stub, une instance de `IServiceProxyAction_Stub`. Nous devons passer l'URL de destination au stub *via* sa méthode `_setTargetEndpoint()`. L'URL est *http://localhost:8080/corpdata/jaxrpc/IServiceProxyAction*, qui est l'emplacement du service final que nous avons déployé plus tôt. Nous pouvons maintenant simplement appeler la méthode `getCotationAction()`, ce qui entraîne l'invocation de la méthode de service de même nom. Remarquez que la valeur de retour est une instance de `javasoap.livre.ch9.services.ProxyCotation`. La sérialisation de ce type personnalisé est prise en charge automatiquement à la fois côté client et serveurs. L'exécution de cette application produit le résultat suivant :

```
Prix : 0.0
Volume : 0
Dernière modif : NA
Différence : 0.0
```

C'est ainsi que vous pourriez avoir à utiliser JAX-RPC pour développer à la fois les codes client et service. Mais ce ne sera pas toujours le cas. J'ai indiqué plus haut que `xrpcc` pouvait utiliser un fichier WSDL. Cela serait une façon élégante de générer les stubs permettant d'accéder par exemple à un service GLUE. (Et ce serait probablement la meilleure façon de générer les stubs permettant d'accéder à un service quelconque sur le réseau.) Nous allons essayer cela.

Générer des stubs par WSDL

Générer le code client par WSDL est assez simple. Nous aurons d'abord à modifier le fichier *CorpDataServices.xml* que nous avons utilisé plus haut comme entrée de `xrpcc`. Nous devons supprimer toute la section `rmi` et la remplacer par une section `wsdl`. Voici à quoi le fichier doit ressembler :

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
  <wsdl name="CorpDataServicesProxy"
        location="http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl"
        packageName="javasoap.livre.ch11.glueclient">
  </wsdl>
</configuration>
```

L'attribut `name` spécifie le nom de modèle comme dans l'exemple RMI précédent. L'attribut `location` contient l'adresse d'une ressource WSDL pour le service. Nous allons utiliser le service GLUE du chapitre 9. Son WSDL est *http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl*. L'attribut `packageName` comme auparavant, spécifie le nom de paquetage du code généré. Cette fois, nous utiliserons `javasoap.livre.ch11.glueclient`, dans la mesure où ce code est destiné à accéder à notre service GLUE. Générez le code client par la commande :

```
xrpcc -client -keep -classpath e:\projects\soap -d gen CorpDataServices.xml
```

Il s'agit globalement de la même commande que précédemment, hormis que l'option `-client` indique que nous ne voulons générer que le code client. Une fois le processus terminé, les fichiers de classes Java doivent être déplacés dans le répertoire java-

soap.livre.ch11.glueclient de façon à apparaître dans le classpath. Notez la classe javasoap.livre.ch11.glueclient.Cotation. Il s'agit d'un type personnalisé représentant les données renvoyées par le service GLUE. Elle est déjà définie en tant que type dans le document WSDL. Parfait ! Nous n'aurons rien à faire de particulier pour la prendre en charge.

Écrivons une application cliente nommée javasoap.livre.ch11.glueclient.AppCotationAction. Elle ressemble à l'autre AppCotationAction, mais elle va communiquer avec notre service GLUE existant. Voici le code :

```
package javasoap.livre.ch11.glueclient;

public class AppCotationAction {
   public static void main(String[] args) throws Exception {
                CorpDataServiceImpl proxy =
          new CorpDataServiceImpl();
                CorpDataServiceSoap_Stub stub =
          (CorpDataServiceSoap_Stub)proxy.getCorpDataServiceSoap();
                stub._setTargetEndpoint(args[0]);
                Quote cotation = stub.getQuote("IBM");
                System.out.println("Prix : " + cotation.get_dernierPrix());
                System.out.println("Volume : " + cotation.get_volume());
                System.out.println("Dernière modif : " + cotation.get_date());
                System.out.println("Différence : " + cotation.get_change());
   }
}
```

Exécuter cette application nécessite que l'URL du service final soit passée dans la ligne de commande. Dans notre cas, l'URL est *http://mindstrm.com:8004/glue/urn:CorpDataServices*. À l'exécution, j'obtiens le résultat suivant :

```
Prix : 105.09
Volume : 10726400
Dernière modif : Mar  8
Différence : 1.38
```

Nous pourrions utiliser le code de cet exemple pour permettre à notre service précédent de communiquer avec le service GLUE pour récupérer ses données au lieu de renvoyer des données factices comme on le fait actuellement. À ce stade, vous devriez entrevoir comment faire cette modification et je passerai donc rapidement là-dessus. Nous devrons modifier la méthode getCotationAction() de la classe ServiceProxyAction. Voici à quoi ressemblerait le code modifié :

```
package javasoap.livre.ch11.services;

import javasoap.livre.ch9.services.ProxyCotation;
import javasoap.livre.ch11.glueclient.*;

public class ServiceProxyAction {

   public ProxyCotation getCotationAction(String symbole) {
```

```
        ProxyCotation pq = new ProxyCotation();
        try {
           CorpDataServiceImpl proxy =
               new CorpDataServiceImpl();
           CorpDataServiceSoap_Stub stub =
               (CorpDataServiceSoap_Stub)proxy.getCorpDataServiceSoap();
           stub._setTargetEndpoint(
               "http://mindstrm.com:8004/glue/urn:CorpDataServices");
           Quote cotation = stub.getQuote(symbole);

           pq.setLast(cotation.get_lastPrice());
           pq.setVol(cotation.get_volume());
           pq.setTime(cotation.get_timeStamp());
           pq.setDiff(cotation.get_change());
        }
        catch (Exception ex) {
           ex.printStackTrace();
        }


        return pq;
     }
  }
```

Le paquetage `javasoap.livre.ch11.glueclient` est importé. Il contient le code du stub pour communiquer avec le service `urn:CorpDataServices`. Nous avons cette fois codé en dur l'URL passée à `_setTargetEndpoint()`. Après avoir récupéré une instance de `Cotation` par l'appel à `getCotation()`, nous l'utilisons pour renseigner une instance de `ProxyCotation`. Souvenez-vous, les deux services renvoient les mêmes types de données. Nous renvoyons donc simplement l'instance de `ProxyCotation`. Ceci ne pose pas de difficulté grâce à WSDL. Naturellement, nous avons à compiler la classe modifiée `ServiceProxyAction` et à créer et déployer un fichier WAR mis à jour pour inclure le code modifié aux côtés de l'ensemble du paquetage `javasoap.livre.ch11.glueclient`.

DII (Dynamic Invocation Interface)

Vous souhaiterez parfois exercer plus de contrôle sur la façon dont les interfaces de vos applications s'interfacent avec un service. L'une des raisons pour cela est que votre application ne connaît pas à l'avance les méthodes et leurs paramètres du service, ce qui rend impossible la génération de stubs. Il est également prévisible que le service que vous voulez invoquer ne fournisse pas de document WSDL le décrivant. La DII (*Dynamic Invocation Interface*) est une API qui vous permet d'invoquer des services à un niveau plus proche du moteur d'exécution JAX-RPC. Nous n'allons pas aborder DII mais vous ne devez pas ignorer son existence. Nous avons eu un aperçu de sa structure au chapitre 9 lorsque nous avons développé un exemple Axis élémentaire. Il peut ne pas être identique et ces API évoluent sans cesse mais l'approche est la même.

Je considère DII comme une approche de dernier recours pour écrire du code communiquant avec des services web SOAP. Laisser un outil générer le code de bas niveau en

se basant sur une description WSDL est vraiment le meilleur choix, Dans la mesure où WSDL est indépendant du langage et de l'implémentation et est un standard accepté. Si vos outils SOAP peuvent utiliser des fichiers WSDL, ce sera la meilleure solution.

JAXM en résumé

Les API fournies par JAXM pour l'écriture de services et de clients sont, à un niveau très bas, similaires à ce que nous avons vu au chapitre 8. Elles nécessitent que vous effectuiez une connexion au serveur (ou à un intermédiaire) et fabriquiez le XML qui constitue le message. Bien qu'il s'agisse là d'une technique puissante, cela vous donne surtout la corde pour vous pendre. J'apprécie la flexibilité de l'emploi direct de XML, mais je veux conserver la structure d'une interaction de type RPC. Si vous avez besoin de concevoir vos services pour qu'ils acceptent des documents XML en paramètre, vous pourrez vous contenter de l'encodage littéral dans le contexte RPC. Si cela n'est pas suffisant, vous devrez alors vous engager sur le chemin des messages.

Je vous suggère de jeter un œil aux exemples du didacticiel Java Web Services Developer Pack. Même si nous n'allons pas ici développer d'exemple JAXM, vous possédez les connaissances suffisantes pour comprendre de quoi il en retourne. Et quelque chose me dit qu'une future édition de ce livre nous plongera dans JAXM, JAX-RPC et le reste du JAX Pack.

Et ensuite ?

Vous êtes maintenant plus que prêt à commencer à développer des services web en Java. Ne vous laissez pas dépasser par les changements. Restez au courant des dernières technologies y compris des dernières versions d'implémentations existantes. De nouvelles versions de GLUE, Axis (Apache SOAP), et d'autres rendront le développement de services en Java plus simple et plus puissant. Enfin, gardez un œil sur les API standardisées, la sécurité, les registres de services et l'intégration dans les environnements de développement intégrés. La liste augmente aussi vite que les besoins en bande passante ! Plongez !

Index

A

accesseurs
  aux valeurs de paramètres des méthodes RPC SOAP 48
  propriétés de classe, modèles de conception de JavaBeans 98
accesseurs polymorphes 27, 28, 36
  pour les tableaux hétérogènes 39
acteurs 163
  attributs actor
    élément faultactor 22
activation de service
  au niveau application 50, 53
  au niveau requête 50
  au niveau session 50
activation framework (JavaBeans), prise en charge des attachements MIME 175
actor, attribut 17
addElement()
  classe Element 153
  classe Parameter 76
addParameter() (Call) 217
addProperty() (Context) 58
addText() 153
adresses, service 193
  service .NET d'intermédiation 212
analyse des flux de données du service web de cotation boursière 187
analyseMessageFormate() 119
Ant 235
Apache SOAP xii, 6
  API de traitement XML 122
  applications clientes 62–66
    appel depuis un service GLUE 193–199
    pour un service d'intermédiation .NET 212
  classe SOAPException, version GLUE vs. 152
  classes fournisseurs et routeurs, remplacement 220–225
  déploiement de services 52–56
    application cliente gestionnaire de service 54
    classe DOMFaultListener 53
    modification du déploiement en niveau requête 70
    outil d'administration 54
  exceptions côté serveur 143–146
  module d'écoute de fautes, création 147–151
  paramètres in, out et in/out, non supportés 81
  passage de tableaux en paramètres d'une méthode de service 86–91
  passage de tableaux en paramètres de méthodes de services
    application cliente 86
    application cliente passant un tableau hétérogène 89
    enveloppe SOAP pour le service getVolumeTotal 87
    tableaux hétérogènes 88

passage de types personnalisés en paramètres de méthodes de services 102–105
déploiement d'un service 102–105
renvoi de tableaux aux appels à une méthode de service 95–96
client (exemple) 95
enveloppe SOAP pour la méthode 96
renvoi de types personnalisés 111–113
application cliente 112
déploiement de ServiceBoursierDeBase 111
sérialisation personnalisée de tableaux épars 126–136
service d'intermédiation utilisant un service basé sur GLUE 199–204
service de messages, écriture 159–168
client 161–164
descripteurs de déploiement 160
enveloppe 161–163
URL pour le routeur SOAP 163
service prenant en charge les en-têtes SOAP 225–231
élément targetAddress dans l'en-tête 226
ServiceProxyCotationRedirige (exemple) 226–231
ServicePrixActions (exemple)
accès 75
déploiement 74
sessions 72
maintient au travers d'appels multiples 73
(voir également Axis) 6
Apache2Glue, application 194–195
Apache2Glue_v2, application 197
API GLUE 79
API Java pour les messages XML (JAXM) xii
API Java pour les RPC XML (voir JAX-RPC) xii
AppAppelleCompteur, classe (exemple) 58
appels de méthodes 2
Apache SOAP 64
classe Call 63
mise en correspondance avec du code Java 47
réponse de méthode 48
réponses de fautes 49
RPC SOAP 45–51
appels de procédures distants (RPC remote procedure calls) xi
appels de procédures distants (voir RPC) 2
appendChild()
classe Document 163
classe Element 163
application Apache2Glue 194–195
application Apache2Glue_v2 197
Application, champ (écran de console GLUE) 59
applications clientes
Apache Axis, pour un service .NET 216–218
Apache SOAP
appel depuis un service GLUE 193–199
document XML, transmission 170
en-têtes SOAP 229
faute et section détails 150
informations sur la température, demande d' 166
passage de tableaux hétérogènes en paramètres de méthodes de services 89
pour passer des types personnalisés en paramètres de méthode 103
pour un service d'intermédiation .NET 212
renvoi de tableaux aux appels de méthode 95
renvoi de types personnalisés 112
sérialisation personnalisée 134
service .NET, accès 214
service de messages RegistreMeteo 161–164
ServiceBoursierDeBase, accès à 86
Apache SOAPServiceSourceDeDonnees, accès à 124
classe personnalisée OrdreBoursier_CoteClient 99
écriture pour les services SOAP de type RPC 62–70
Apache SOAP 62–66
GLUE 66–70
fautes SOAP 21

GLUE
appel d'un service Apache SOAP depuis 204–218
exception, renvoi avec détails 153
passage de types personnalisés en paramètres 106–109
renvoi de tableaux aux appels à des méthodes de service 96
renvoi de types personnalisés 114
tables de hachage, sérialisation personnalisée 141
wsdl2java et génération de classes Java pour les types personnalisés 107
JAX-RPC
création 242
pour un service GLUE 243–245
AppServiceSourceDonnees, classe (exemple) 140
AppSession, classe (exemple) 72
arbres, racine ou point de départ 43
Array, classe 130, 131, 136
ArrayList, classe 218
ArraySerializer, classe 126
attachements à des messages SOAP 158
données binaires 158
écriture du contenu dans un fichier local 182
encodage littéral et 173–182
type MIME, prise en charge par le JavaBeans Activation Framework 175
ATTR_MUST_UNDERSTAND, constante 231
attribut actor 17
attribut defaultRegistryClass (isd:élément mappings) 133
attribut encodingStyle 19, 65
applications GLUE 79
élément Envelope 16
élément getPrix (ServicePrixActions) 77
attribut href
faisant référence aux valeurs d'autres éléments 28
attribut id 28
attribut java2XMLClassName 103, 124
attribut javaType 103
attribut methods (élément provider) 53
attribut mustUnderstand 18, 21, 226
dans les en-têtes SOAP 231
élément targetAddress, positionnement pour l' 231
attribut ns2:arrayType (éléments de tableaux hétérogènes) 91
attribut packageName
paquetage Java utilisé côté client 238
attribut position (pour les éléments d'un tableau) 130
attribut root 43
attribut scope (élément provider) 53, 70
attribut servantName 238
attribut targetNamespace 238
attribut type
élément portType 192
élément provider 53
isd:élément service 161
attribut typeNamespace 238
attribut xml2JavaClassName 103, 124
attribut xmlns
positionnement pour un service de messages 163
attributs XML
actor 17
attribut arrayType 38, 39
attribut encodingStyle 65
applications GLUE 79
attribut href 28
attribut id 28
attribut java2XMLClassName 103
attribut javaType 103
attribut location 195
attribut offset 41
attribut packageName 238
attribut parameterOrder 192
attribut root 43
attribut servantName 238
attribut soapAction 192
attribut type 53, 161, 192
attribut xml2JavaClassName 103
déclarations d'identifiants d'espaces de noms 26
encodingStyle 16, 19
espace de noms
positionnement pour un service de messages 163
qualificateurs pour 16

mustUnderstand 18, 231
erreurs associée à 21
pour élément Header 17
pour l'élément provider 53
attributs methods 53
sérialisation 130
targetNamespace 191, 204, 238
typeNamespace 238
AutreServiceFaute, classe (exemple) 152
AutreServiceSourceDonnees, classe (exemple) 139
Axis
API JAXM et JAX-RPC xii, 6
client pour un service .NET, écriture d'un 216–218

B

balises d'éléments, document WSDL 191
balises documentation, documents WSDL 193
Base 96
base64, jeu de caractères 34
Bean, classe 123
BeanSerializer, classe 104, 113, 197
bind() (Registry) 67
Body, élément 13, 20
style d'encodage XML littéral, positionnement 171
ByteArrayDataSource, classe 176

C

Call, classe 63, 76, 104, 231
setTargetObjectURI() 193
caractères spéciaux, séquences d'échappement 33
casse, conversion du symbole de l'action 81
chaînes 33
Java et SOAP 27
retours d'appels de méthodes RPC Apache SOAP, conversion en 64
tableaux de
passage en paramètres de services 87
valeurs de retour d'une méthode de service 95
chaînes de caracères
envoi comme paramètres aux méthodes des services 76
champ Application (écran de console GLUE) 59
champ d'en-tête SOAP Action 10
interopérabilité, problèmes avec le 183
services .NET
paramètrage 212
positionnement 217
champ HOME (écran de console GLUE) 59
chargement dynamique de classes, implémentations SOAP Java 52
cherche() (DataHandler) 175
classe AppAppelleCompteur (exemple) 58
classe AppServiceSourceDonnees (exemple) 140
classe AppSession (exemple) 72
classe Array 136
getLength() 130
set() 131
classe ArraySerializer 126
classe AutreServiceFaute (exemple)
envoi d'exception détaillées 152
classe AutreServiceSourceDonnees (exemple) 139
classe Bean 123
classe BeanSerializer 104, 113
classe ByteArrayDataSource 176
classe Call 63, 217
correspondances de types personnalisés avec des classes Java 104
paramètres de méthode, positionnement 76
setHeader() 231
URI du style d'encodage, positionnement 76
classe CompteurMethode (exemple) 52
classe Constants 76, 104
classe Context (GLUE) 58
classe Cotation (exemple) 186
description d'un document WSDL 195
classe DataHandler 175
classe DataSource 176

classe Document 162
classe DocumentBuilder 162, 169
classe DOM2Writer 171
classe DOMFaultListener 53
classe DOMUtils 122
classe Element 122
classe Envelope 5
  unmarshall() 163
classe Exception 144
classe ExceptionSourceDonnees (exemple) 147
classe Fault 144
classe File 177
classe FileReader 169
classe FournisseurJavaRPCAmeliore (exemple) 221
classe Hashtable 119, 139, 147
classe HashtableType 137–138
classe Header 222, 231
classe HeaderClient (exemple) 229
classe HTTP
  démarrage d'une instance d'un serveur 58
  expiration de session, paramétrage 74
classe InputSource 167, 169
classe Java
  classe CompteurMethode (exemple), dans ServiceCompteurAppels 51
  implémentation de services SOAP
    noms de méthodes 53
classe Java Axis2DotNet 216
classe Java Axis2Glue 218
classe Liste 173, 182
classe Mappings 138
classe Message
  création d'une instance et appel de la méthode send() 163
classe ModEcouteFauteSrcDonnees (exemple) 148
classe Parameter 63, 76
classe personnalisée
  MinMax_CoteClient 109, 114
  OdreBoursier_CoteClient 99, 100
    générée par wsdl2java 108
classe PrixActions (exemple) 74
  paramètres in/out dans les méthodes 81
classe Qname 105
classe RegistreCorrespSOAPAmeliore (exemple) 133
classe Registry (GLUE)
  liaison d'une interface de Java à un service SOAP 67
  publication d'un service 58
classe RemoteException 236
classe Reponse 63
classe Response
  getReturnValue() 144
classe RouteurRPCAmeliore (exemple) 222–225
classe RPCJavaProvider 220, 221
classe RPCRouter 220
  invoke() 222
classe RPCRouterServlet 220
classe SerialiseurTableauEpars (exemple) 126–131
classe Service 217
classe ServiceBoursierDeBase (exemple) 85
  client Apache SOAP 86
  déploiement sous Apache SOAP 86
  méthode de service executeOrdreBoursier() 101
  méthode de service getLaPlusActive 94
  méthode de service getMinMax() 111
  renvoi de tableaux aux appels à des méthodes de service
    GLUE, déploiement sous 96
  renvoi de tableaux aux appels de méthode 94–98
  tableau hétérogène en paramètre de méthode 88
classe ServiceBoursierDeBaseHelper (exemple) 108
classe ServiceFauteExemple (exemple) 144
  renvoi d'une exception avec information détaillée 148
classe ServiceProxyAction (exemple) 237
classe ServiceProxyActionImpl (exemple) 236
classe SOAPContext 159, 222
classe SoapEncUtils 122
classe SOAPException 145
  implémentation GLUE 152

récupération du code de faute, du libellé et de l'acteur 154
classe SOAPFaultEvent 147
classe SOAPHTTPConnection 72
classe SOAPMappingRegistry 104, 133
classe SOAPTransport 163
classe StringWriter 130
classe Type 137
classe Value 139
classe Vector 76
classe Writer 122
classe XMLParserUtils 149
getXMLDocBuilder() 162
classes
routeur et fournisseur Apache SOAP, remplacement 220–225
classes Java
Apache SOAP fournisseurs et routeurs 220
association à des objets Java 123
chargement dynamique en SOAP 52
classe Call 63
classe Vector 76
conception 98–100
côté client, pour accéder à un service GLUE 61
désérialisation de données encodées dans des instances de 122
Hashtable 119
partage en programmation client/serveur 106
personnalisée, sérialisation d'objet 120
porteuses 81
structure de données 107
classe MinMax_CoteServeur 114
correspondance avec les champs des types personnalisés 108
structures vs. 37
techniques, génération sous la console GLUE 68
classes porteuses, encapsulation de types de données pour les paramètres in/out et out 81
classes techniques
classe ServiceBoursierDeBaseHelper (exemple) 108
génération par wsdl2java 107
génération sous la console GLUE 68
client gestionnaire de service (Apache SOAP) 54
déploiement ServicePrixActions (exemple) 75
suppression et redéploiement de services 70
codepostal élément, création pour le service de messages RegistreMeteo 163
collections, sérialisation 137–142
COM (Component Object Model)
polymorphisme dans le type variant 36
complexContent, élément
document WSDL 191
composants logiciels pour les exemples 8
CompteurMethode, classe (exemple) 52
console, GLUE 58
code Java, génération pour l'accès à un service 68
constantes
ATTR_MUST_UNDERSTAND 231
NS_PRE_SOAP_ENV 231
NS_URI_LITERAL_XML 171
NS_URI_SOAP_ENC 76, 104, 171
NS_URI_SOAP_ENV 162
Constants, classe 76, 104
contenu
langages à balises, mise en forme avec 4
spécifications de type transmission 5
Context, classe (GLUE) 58
contexte de réponse, racine 160
cookies 65
sessions, maintenance avec des 72
CORBA 2
CorpDataServices (exemple) 185–193
accès par un client Axis 218
classe CorpDataService 187
classe CorpDataServices 187
getCotation() 185
getHeadLines() 185
recherche du document WSDL associé aux services publiés 188–190
service .NET d'intermédiation 211
service d'intermédiation utilisant Apache SOAP 199–204
service GLUE getHeadLines, appel depuis un client Apache 193
type personnalisé Cotation 185

corps (messages SOAP), récupération d'éléments dans le 160
correspondance
champs d'un type personnalisé vers une classe Java contenant une structure de données 107
de types personnalisés avec des classes Java 102, 108
valeurs de retour d'une méthode de service vers un type de données personnalisé 111
corrigeSymbole() (PrixActions), méthode de service 81
Cotation 200
Cotation, classe (exemple) 191
côté client (Apache), code 196
Cotation, exemple de classe 186
CotationParser, classe (exemple) 187
createElement() 149, 163
createElementNS() 162

D

DataHandler, classe 175
DataServices 193
DataSource, classe 176
déclarations
identifiants d'espaces de noms 26
tableau, unidimensionnel 38
type d'élément chaîne 33
type, le polymorphisme SOAP et les 36
defaultRegistryClass (isd:élément mappings), attribut 133
definitions, élément 185, 191
attribut targetNamespace 204
déploiement
service comprenant un type personnalisé en paramètre de méthode 102–105
service JAX-RPC sur un serveur Tomcat 241
service RegistreMeteo 160
ServiceBoursierDeBase
ajout de la méthode de service getMinMax 111
sous Apache SOAP 86
ServicePrixActions, sous Apache SOAP 74
ServiceProxyCotationRedirige (exemple) 228
services avec une portée de niveau requête 70–71
services avec une portée de niveau session 72–74
services SOAP de type RPC 52–62
sous Apache SOAP 52–56
sous GLUE 57–62
ServiceSourceDeDonnees 123
registre de mises en correspondance, ajout 133
descripteurs de déploiement 52–54
pour un service JAX-RPC 239
description de services web (voir WSDL) 184
désérialisation
(voir aussi sérialisation) 139
données encodées dans une instance de classe Java 122
tableaux épars 131
types personnalisés 139
Deserializer, interface 120
destinations finales
service proxy 231
detail, élément 23, 146
DII (Dynamic Invocation Interface) 245
Document, classe 162
documentation, balises
documents WSDL 193
DocumentBuilder, classe 162, 169
documents XML
enveloppe du message de réponse, construction à partir de 167
envoi dans la réponse d'un service de messages 164–166
transmission par SOAP 157
documents, WSDL 185
associé aux services publiés, recherche 188–190
génération par l'utilitaire java2wsdl 204–208
DOM2Writer, classe 171
DOMFaultListener, classe 53
DOMUtils, classe 122, 131
données autodescriptives 2
HTML et XML 4

données binaires
encodage 34–36
envoi sous forme d'attachements SOAP 158
données d'authentification, stockage dans l'en-tête SOAP 219
Dynamic Invocation Interface (DII) 245

E

echoActions() (ServiceSourceDeDonnees) 132
écriture de clients pour les services SOAP de type RPC 62–70
Apache SOAP 62–66
élément Body 13, 20
faute sous-éléments 21–23
élément detail 23
élément Envelope 13, 16
(voir également enveloppes, SOAP) 13
attribut encodingStyle 16
élément Header 16, 231
attribut mustUnderstand 18
espaces de noms qualificateurs pour 16
élément java 53
élément mappings 53
élément provider 53
attribut scope, modification 70
élément java 53
élément racine dans les documents XML
récupération 169
Element, classe 122
éléments d'un service 53
élément faultListener 53
élément provider 53
éléments de données pour les types personnalisés 207
éléments de types 191
éléments detail 146
création pour les fautes GLUE 152
extraction d'information par l'application cliente 150
récupération pour les fautes 154
éléments enfants, récupération pour l'élément racine 168
éléments operation, documents WSDL 192
éléments racine de documents XML 168
élément definitions 185, 191
éléments vides, accédant aux valeurs d'autres éléments 28
éléments XML
encapsulés dans les messages SOAP 28
indépendants dans les messages SOAP 28
références à des valeurs d'autres éléments 28
service 193
éléments, tableaux 38
tableaux multidimensionnel 38
encodage de données 25–44
attribut root 43
encodage littéral 157
encodage littéral (XML) 168–182
attachements 173–182
références à des valeurs d'autres éléments 28
règles de sérialisation 28–31
schémas et espaces de noms 25–28
terminologie 26–28
type, indication 31–42
accesseurs polymorphes 36
encodage binaire 34–36
types composés 37–41
types simples 32–34
valeurs par défaut 42
encodage littéral 168–182
attachements 173–182
encoder des données (voir aussi encodage de données) 25–44
encodingStyle, attribut 19, 65
en-têtes
HTTP
réponse du serveur aux requêtes du client Apache SOAP 65
requêtes 10
SOAP 13, 219–231
données optionnelles 50
mise à disposition des méthodes de service Apache 220–225
service prenant en charge les, implémentation 225–231
SOAPAction 10
entiers
énumérations du langage C et 33
Java et SOAP 27

énumérations 33
Envelope, classe 5, 163
Envelope, élément 13, 16
enveloppes, SOAP 9, 13–16
  Apache SOAP
    méthode de service echoActions 136
    méthode de service renvoyant un type personnalisé 113
    passage de tableaux en paramètres de méthodes de services 87
  appel de méthode Java, encapsulation 52
  appels de méthodes RPC, mise en correspondance de Java avec les 48
  en-tête SOAP dans l' 231
  espaces de noms XML 13
  fautes dans les espaces de noms 21
  gestion des versions 20
  GLUE 79
    méthode de service envoieMessage et table de hachage sérialisée 141
    méthode de service renvoyant des types personnalisés 115
    passage de types personnalisés en paramètres 109
    réponse du serveur 80
  invocation de la méthode de service executeOrdre() 90
  obtention par la méthode locate() du routeur par défaut 222
  pour l'appel de la méthode de service executeOrdre 93
  service de messages 159
    conversion d'un document XML en une instance d'enveloppe 163
  service RegistreMeteo, construction pour le 161–163
  service transmettant des documents XML 171
  utilisation des paramètres out et in/out, effets sur les 83
envoieMessage(), méthode de service
  interface IAutreServiceSourceDonnees 140
  table de hachage en paramètre 139
erreurs
  fautes SOAP 21–23
    élément detail 23
    élément Fault 21
    élément faultactor 22
    élément faultcode 21, 146
    élément faultstring 22, 146
    exemple de 23
  SOAP versions d'enveloppe dans un espace de noms 20
erreurs, codes HTTP 12
espaces de noms par défaut, XML 15
espaces de noms, XML
  client Apache SOAP pour un service .NET d'intermédiation 213
  comme propriété de l'objet Call 217
  défaut 15
  définition de service WSDL 191, 238
  encodage de données SOAP 25–28
    constante Apache SOAP pour le style d'encodage 76
  enveloppes, fautes dans 21
  identifiant d'espace de noms ns2 88
  identifiants 14, 26
    enveloppe de message SOAP 162
  implémentation GLUE de l'encodage des tables de hachage 142
  méthodes de service d'une application GLUE 79
  pile de définitions (pileEN) 122
  pour WSDL lui-même 191
  qualificateurs pour
    attribut encodingStyle 19
    création pour des éléments 162
    élément Envelope 16
    élément Header 16
    exceptions et 154
    méthode RPC SOAP invoquée 46
    service déployé avec Apache SOAP 53
    type personnalisé 102
  SOAP versions d'enveloppe 20
  (voir également SOAP-ENV) 21
Exception, classe 144
exceptions
  classe ExceptionSourceDonnees (exemple) 147
  classe SOAPException 145
  sérialisation 121

sous GLUE 152–155
application cliente 153
classe SOAPException 152
exceptions côté serveur, Apache SOAP 143–146
ExceptionSourceDonnees, classe (exemple) 147
executeOrdre(), méthode de service 93
executeOrdreBoursier(), méthode de service 101, 106
exemples de ce livre, téléchargement xv
extensibilité
de SOAP 5
XML vs. HTML 4

F

Fault, classe 144
faultGenerated() (Response) 144
faute() (SOAPFaultListener) 147, 148
fautes 21–23, 143–155
appels de méthodes RPC SOAP 49
classe ServiceFauteExemple (exemple) 144
élément detail 23
élément Fault 21
élément faultactor 22
élément faultcode 21, 146
élément faultListener 53
élément faultstring 22, 146
exceptions côté serveur, Apache SOAP 143–146
exceptions sous GLUE 152–155
application cliente 153
classe SOAPException 152
exemple de 23
module d'écoute de fautes, création sous Apache SOAP 147–151
service de messages, Apache 160
fichier de configuration pour xrpcc 237
fichier image, attaché à l'invocation d'une méthode de service 175–182
fichier WAR (web application archive) 240
File, classe 177
FileReader, classe 169
format de données transmises (voir aussi spécifications de niveau transmission) 6
fournisseur connectable (Apache SOAP) 220
FournisseurJavaRPCAmeliore, classe (exemple) 221
fournisseurs, Apache SOAP 220
remplacement de classes 220–225
classe FournisseurJavaRPC-Ameliore (exemple) 221

G

generatedFault() (Response) 151
generateNullStructure() (SoapEncUtils) 122
generateNullStructure(), classe (SoapEnc-Utils) 122
generateStructureHeader() (SoapEnc-Utils) 122
genereFaute() 153
classe ServiceFauteExemple (exemple) 144
renvoi d'une instance de SOAP-Exception 146
getAttributeNS (DOMUtils) 131
getBody() 160
getBodyEntries() 160
getChildCharacterData() (classe DOM-Utils) 122
getCotation(), méthode de service
classe ProxyCotationRedirige 228
implémentation d'un service .NET d'intermédiation 211
getCotationAction()
classe ProxyCotation (exemple) 236
classe ServiceProxyAction (exemple) 237
classe ServiceProxyActionImpl (exemple) 237
getCotations(), méthode de service
classe ServiceProxyCotation 202
getDataSource() (DataSource) 176
getDetailEntries() (Faute) 151
getDocumentElement() 169
getElementsByTagName() 168
getFault() (Response) 144
getFirstChild() (Element) 151
getHeader() (Envelope) 222
getHeadLines(), méthode de service
CorpDataServices 185
getImage() 177, 182

getLaPlusActive(), méthode de service 94, 96
getLength() 130
getListe() 177
getMessage() (SOAPException) 154
getMinMax(), méthode de service 111, 114
getObject() 139
getPrix() (ServicePrixActions), méthode de service 74
getPrixOut() (PrixActions), méthode de service 81
  CorpDataServices 185
  service GLUE, appel depuis un client Apache SOAP 195
getReaders() (Reader) 139
getReturnValue() (Response) 63, 144
getSOAPActor() (SOAPException) 154
getSOAPCode() (SOAPException) 154
getSOAPDetailElement() 154
getSOAPException() 149
getSOAPTransport() (Message) 163
getTagName() 160
getTargetAddress() 228
getTemperature(), méthode de service (RegistreMeteo) 164–166
getValue() (Parameter) 63
getVolumeTotal(), méthode de service 87, 92
  ServiceBoursierDeBase 86
getXMLDocBuilder() (XMLParserUtils) 149, 162
GLUE 7
  client et serveur, partage de classes et d'interfaces 106
  clients
    appel d'un service Apache SOAP depuis 204–218
    pour les services SOAP de type RPC 66–70
  déploiement de services sous 57–62
    console GLUE 58
  exceptions 152–155
    application cliente 153
    classe SOAPException 152
    élément detail, création 152
  gestion des sessions 74
  JAX-RPC application cliente pour un service 243–245
  messages SOAP transmis par des applications 79
  paramètres de méthode
    passage de in, out et in/out 80–84
    passage de tableaux en 91–94
    passage de types personnalisés en 105–109
  passage de paramètres de méthode 77–80
  renvoi de tableaux aux appels à des méthodes de service 96
    client, écriture d'un 96
    enveloppe SOAP pour getLaPlusActive() 97
  renvoi de types personnalisés 113–115
    application cliente 114
  sérialisation personnalisée 137–142
  services
    accès par un client Axis 218
    appel depuis un client Apache SOAP 193–199
    service d'intermédiation utilisant Apache 199–204
graphes orientés, racine ou point de départ 43

H

Hashtable, classe 119, 139, 147
HashtableType, classe 137–138
Header, classe 222, 231
Header, élément 16, 231
  HTTP
    réponse 11
HeaderClient, classe (exemple) 229
HeadlineParser, classe (exemple) 187
HOME, champ (écran de console GLUE) 59
hôtes, HTTP champs d'en-tête pour 10
href, attribut 28
HTML, données autodescriptives 4
HTTP 5
  connexions, maintenance des sessions dans 72
  informations de type cookie dans les en-têtes 50
  réponse 11
  requêtes 10
  système de messages SOAP 159

systèmes SOAP basés sur, activation 50
transport des services de messages 163
HTTP, classe 58, 74

I

IAutreServiceSourceDonnees, interface (exemple) 140
id, attribut 28
identifiant d'espace de noms
isd 53
ns2 93
xsd 26
xsi 26
identifiants
élément XML, espace de noms 14
SOAP-ENV 16
enveloppe SOAP générée sous GLUE 79
espace de noms des éléments XML
enveloppe de message SOAP 162
espaces de noms d'éléments XML
isd 53
identifiants de listes (ServiceListeVoituresOccasion) 181
XML espace de noms d'élément
XML Schema Part 2 26
implémentations
SOAP 6
Apache 6
Axis 7
GLUE 7
Microsoft .NET 7
SOAP et 5
information de routage dans les en-têtes SOAP 219
informatique distribuée 1
RPC et systèmes orientés messages 2
sécurité, SOAP et 7
input, élément 192
opérations WSDL 193
InputSource, classe 167, 169
interface de service, implémentations des classes 238
interface Deserializer 120
interface IAutreServiceSourceDonnees (exemple) 140
interface IReader (exemple) 139
interface IServiceBoursierDeBase (exemple) 91
méthode de service executeOrdreBoursier() 105
méthode de service getLaPlusActive() 96
méthode de service getMinMax() 114
interface IServicePrixActions, méthodes utilisant des paramètres in/out 82
interface IServiceProxyAction (exemple) 236
interface Provider 221
interface Remote 236
interface Serializable 117
interface Serializer 120
interface SOAPFaultListener 147
interface, balise 238
interfaces Java
liaison aux services
génération avec wsdl2java 107
implémentation GLUE 77
programmation client/serveur programmation, partage en 106
Serializable 117
vers des services SOAP de type RPC 66
interopérabilité, SOAP 62
Microsoft .NET et les services écrits en Java 210–216
WSDL et 183–218
appel d'un service Apache SOAP depuis un client GLUE 204–218
appel d'un service GLUE depuis un client Apache 193–199
service d'intermédiation utilisant Apache SOAP 199–204
vue d'ensemble de WSDL 184–193
interopérabilité, spécifications de type API vs. de type transmission 4
invoke()
classe Call 63, 218
classe RouteurRPCAmeliore (exemple) 222, 225
classe RPCRouter 222
interface Provider 221
IReader, interface (exemple) 139
isd:élément map 102
isd:élément mappings 133

isd:élément service, attribut de message type 161
IServiceBoursierDeBase, interface (exemple) 91, 96, 105, 114
IServiceProxyAction, interface (exemple) 236
IWriter, interface (exemple) 139

J

JAF (JavaBeans Activation Framework), prise en charge des attachements MIME 175
JAR (Java Archive) fichiers à inclure dans le classpath 235
Java
  clients pour les services SOAP de type RPC, écriture 62–70
    Apache SOAP 62–66
    GLUE 66–70
  code client pour accéder à un service GLUE 184
  conversion entre XML et Java 113
  liste chaînée avec ses nœuds (exemple) 43
  mécanisme de réflexion 99
  messages basés SOAP et API RPC (JAXM et JAX-RPC) 6
  mise en correspondance de code avec un appel de méthode RPC SOAP 47
  objets
    appels RPC Apache SOAP, renvoi 64
    association à des classes Java 123
  SOAP, utilisation avec 5
  spécifications de niveau API 4
  types de données
    types SOAP correspondants 27
    valeurs composées 27
Java Apache2DotNet, classe 215
Java API for XML (JAX) 233
Java API for XML Messaging (JAXM) 6, 233, 246
Java Axis2DotNet, classe 216
Java Axis2Glue, classe 218
java, élément 53
Java Message Service (JMS) 4
Java RMI (remote method invocation) 2, 236
Java Web Services Developer Pack (Java WSDP) 233
Java, classe (voir classes Java) 51, 53
java2wsdl, utilitaire 204–208
java2XMLClassName, attribut 103, 124
JavaBeans
  conventions de nommage 51
  design modèles, expression des propriétés d'une classe Java 98
JavaBeans Activation Framework (JAF), prise en charge des attachements MIME 175
javaType, attribut 103
JAX (Java API for XML) 233
JAXM (Java API for XML Messaging) 6, 233, 246
JAX-RPC (Java API for XML-based RPC) 6, 234–246
  Ant, utilisation de 235
  client, création 242
  clients stubs, génération à partir de WSDL 243–245
  Dynamic Invocation Interface (DII) 245
  service, création 236–242
jeu de caractères base64 34
JMS (Java Message Service) 4

K

keys() (Hashtable) 139

L

l2java 184
langage à balises extensible (voir XML) 4
langages à balises 4
langages C/C++
  structures 37
  valeurs par défaut pour les paramètres de méthodes en C++ 42
liaisons
  services GLUE 204
  SOAP 204
  WSDL 185, 192
    combinaison avec une adresse physique dans l'élément port 193
    éléments operation pour le service 192

Liste, classe 173, 182
listes chaînées, racine ou point de départ 43
listes de paramètres de longueur variable 88
locate() (Provider) 221
location, attribut (soap:élément address) 195
logiciel middleware orienté messages (message-oriented middleware ou MOM) 4
longueur des tableaux 130

M

M2Writer 171
main(), classe exemple AppAppelle-Compteur 58
Mappings, classe 138
mappings, élément 53
mapTypes() (SOAPMappingRegistry) 104, 197
marshall() 120
  classe xjmr 130
  extension de la portée d'un espace de noms et 122
  sérialisation de tableaux épars 130
Message, classe 163
message, éléments
  document WSDL 191
messagerie, JAXM (Java API for XML Messaging) 6
messages requête/réponse 9, 163, 164–168
  enveloppe réponse, construction d'un document XML à partir de 167
  réponse transmettant des documents XML 172
  RPC SOAP 45–51
    client Apache SOAP 63
    nom de la méthode invoquée 46
    paramètres de méthodes 47
    réponse aux méthodes utilisant des paramètres in/out 84
    réponse de faute 49
    réponse de méthode 48
    réponse du serveur GLUE 80
    requête Apache SOAP 75
    requête GLUE 78
    URI de l'objet cible 45
  service .NET d'intermédiation 213
  SOAP
    dans l'élément Body 20
    HTTP requêtes POST 10
    réponse HTTP 11
messages, JAXM (Java API for XML Messaging) 233, 246
messages, SOAP 9–23
  attribut actor 17
  attribut encodingStyle 19
  attribut mustUnderstand 18
  élément Body 20
  élément Envelope 16
  élément Header 16
  enveloppes 13–16
  fautes 21–23
  liaison HTTP 9
  réponse HTTP 11
  requêtes HTTP 10
  transmis par des applications GLUE 79
  versions d'enveloppe 20
méthode de service corrigeSymbole() (PrixActions) 81
méthode de service enregistreTemperature (RegistreMeteo) 158, 166
méthode de service envoieMessage()
  classe MessageSourceDeDonnees 118
méthode de service executeOrdre()
  enveloppe SOAP pour la 93
méthode de service executeOrdreBoursier()
  classe ServiceBoursierDeBase 101, 106
  interface IServiceBoursierDeBase-Interface 106
méthode de service getCotation()
  implémentation d'un service .NET d'intermédiation 211
méthode de service getCotations()
  classe ServiceProxyCotation 202
méthode de service getDonneesSymbole() (ServiceDonneesEntreprise) 169
méthode de service getHeadLines()
  CorpDataServices 193
  implémentation d'un service .NET d'intermédiation 211
    invocation à partir d'un client Apache Axis 216

méthode de service getLaPlusActive()
interface IServiceBoursierDeBase 96
ServiceBoursierDeBase 94
méthode de service getMinMax()
classe ServiceBoursierDeBase 111
interface IServiceBoursierDeBase 114
méthode de service getPrix() (ServicePrixActions) 74
méthode de service getPrixOut() (PrixActions) 81
méthode de service getVolumeTotal()
enveloppe SOAP pour la 87
enveloppe SOAP, implémentation GLUE 92
méthode de service web ajoutListe() 175
méthodes
C++, définition de valeurs de paramètres par défaut 42
service, exposition sous GLUE 58
methods (élément provider), attribut 53
Microsoft
.NET 7, 210–216
COM (Component Object Model) 2
type variant, polymorphisme dans le 36
Mind Electric, The 7
MinMax_CoteClient, classe personnalisée 109
MinMax_CoteServeur, classe personnalisée 109, 114
mise en correspondance
de types personnalisés et de sérialiseurs sous GLUE 138
des valeurs de retour d'une méthode de service avec des types personnalisés 197
tableau d'éléments primitifs avec tableau de types Java primitifs, mise en correspondance avec un tableau d'éléments primitifs 182
types de données personnalisés, prise en charge par les stubs et les ties 242
valeurs de retour d'une méthode de service avec des types personnalisés 203
CotationProxyService.map (exemple) 208
ModEcouteFauteSrcDonnees, classe (exemple) 148
modèles d'activation de services 50, 71
attribut scope, spécification avec 53
GLUE, spécification sous 58
portée de niveau requête, déploiement avec une 70–71
systèmes SOAP basés sur HTTP 50
termes alternatifs pour les 71
modèles d'activation des services
portée application, ServiceVoituresOccasion (exemple) 179
MOM (message-oriented middleware) 4
mot-clé xmlns 14
mustUnderstand, attribut 18, 21, 226, 231

N

navigateurs, affichage de documents WSDL 188
.NET
Microsoft 210, 216
.NET (implémentation SOAP) 7, 210, 216
client Apache SOAP accédant au service 214–218
newDocument() (DocumentBuilder) 162
newInstance(), Classe 52
nodeToString() (DOM2Writer) 171
nœuds 5
liste chaînée (code Java) 43
noms
conventions de nommage JavaBeans 51
définitions de services en WSDL 191
qualifiés 162
noms pleinement qualifiés 102, 105
type de données personnalisé 203
noms qualifiés 162
NS_PRE_SOAP_ENV, constante 231
NS_URI_LITERAL_XML, constante 171
NS_URI_SOAP_ENC, constante 104, 171
NS_URI_SOAP_ENV, constante 162
ns2:arrayType (éléments de tableaux hétérogènes), attribut 91

O

Object 139
objets, Java
association à des classes Java 123
renvoi par les appels RPC Apache SOAP 64

offset, attribut XML 41
operation, éléments, documents WSDL 192
OrdreBoursier_CoteClient, classe personnalisée 99
OrdreBoursier_CoteServeur, classe personnalisée 100, 108
outil d'administration, Apache SOAP 54
outil xrpcc, génération de stubs et de ties de service 237–240
  service GLUE, stubs pour un 243–245
output, élément 192
  opérations WSDL 193

P

packageName, attribut 238
paires clé/valeur 139
Parameter, classe 63, 76
paramètre réceptacle 122
paramètres de méthode
  appels RPC SOAP, dans les structures d'invocation de méthode 47
  attribut parameterOrder 192
  passage aux services 74–84
    avec l'API GLUE 77–80
    paramètres in, out et in/out 80–84
  passage de tableaux en 85–94
    sous GLUE 91–94
  passage de types personnalisés en 98–109
    correspondance de types personnalisés avec des classes Java 102
  représentation dans un document WSDL par les éléments part 191
  utilitaire java2wsdl 204
paramètres de méthode out 80–84, 191
paramètres in, out et in/out 80–84
  Apache SOAP 81
  paramètre in (Call.PARAM_MODE_IN) 217
  paramètres in/out 81
    documents WSDL 192
paramètres in/out 191
parse() (DocumentBuilder) 168, 169
part, éléments
  documents WSDL 191
persistance, services Apache SOAP 54
pileEN objects 122
points de destination
  réseau, pour les définitions de services WSDL 185
  services Apache SOAP de type RPC 204
port, élément
  document WSDL 193
portée d'un service (voir modèles d'activation de services) 50
portée de niveau application 70
  ServicePrixActions (exemple) 74
portée de niveau requête 70–71
portée de niveau session 72–74
portées 71
  espace de noms, extension 122
  niveau requête, déploiement de services avec un 70–71
  niveau session, déploiement de services avec des 72–74
ports
  console GLUE, spécifications pour la 59
  points de destinations WSDL 185
portType, élément
  documents WSDL 192
position (pour les éléments d'un tableau), attribut 130
POST requêtes, HTTP 10
println() 64
  affichage des résultats du service de liste 182
PrixActions, classe (exemple) 74, 81
programmation client/serveur
  classe personnalisée MinMax_CoteClient 110
  classe personnalisée MinMax_CoteServeur 109
  partage d'interfaces Java 106
properties
  objet Call 217
propriétés 98
  classe personnalisée OrdreBoursier_CoteClient 99
  contexte GLUE 58
protocoles de transport 5
  attribut transport, document WSDL 192
  champ SOAPAction, problèmes avec le 183

service de messages SOAP 163
SOAP, découplage de 5
Provider, interface 221
ProxyCotation, classe (côté client) 208
publish() (Registry) 58
pushScope() 122
put() (Hashtable) 139

Q

Qname, classe 105
qualificateurs, espaces de noms XML
attribut encodingStyle 19
élément Body 21
élément Envelope 16
méthode RPC SOAP invoquée 46
méthodes de service d'une application GLUE 79
pour les types personnalisés 102
services déployés sous Apache SOAP 53
SOAP-ENC 25
sous-éléments et attributs 16

R

racine du contexte de réponse 160
readObject() 137, 139
receive(), transport des services de messages 164
receptacle, écriture dans le 130
recupIDListes() 177
redéploiement de services, Apache SOAP 70
références
à un tableau sérialisé séparément
comme paramètre de méthode de service 93
comme valeur de retour d'une méthode de retour 97
au package pour l'implémentation de la classe côté serveur 109
aux définitions des éléments de données 25
aux valeurs de données d'autres éléments 29, 28
réflexion 99
registre de mise en correspondance 133
RegistreCorrespSOAPAmeliore, classe (exemple) 133
RegistreMeteo, classe (exemple) 159
registres
classe RegistreCorrespSOAPAmeliore (exemple) 133
classe SOAPMappingRegistry 104, 197
mise en correspondance 133
Registry, classe (GLUE) 58, 67
Remote, interface 236
RemoteCotation, classe (exemple) 201
RemoteException, classe 236
répertoire racine pour les applications web (WEB-INF) 240
Reponse, classe 63
requêtes POST, HTTP
mécanisme de requête/réponse 12
requêtes sur une base de données, types de données résultant de 41
Response, classe 144
RMI (remote method invocation) 2, 236
rmi, balise 238
root, attribut 43
root, attribut XML 43
RouteurRPCAmeliore, classe (exemple) 222–225
routeurs, Apache SOAP RPC 54, 193, 204, 220
remplacement de classes 220–225
classe RouteurRPCAmeliore (exemple) 222–225
implémentation de vos propres 222
RPC
appel de méthode, représentation dans SOAP 45–51
HTTP comme transport pour 5
JAX-RPC 6, 234–246
client, création 242
clients stubs, génération à partir de WSDL 243–245
Dynamic Invocation Interface (DII) 245
outil Ant, utilisation de 235
service, création 236–242
passage de paramètres 74–84
systèmes de messages basés sur 9
RPCJavaProvider, classe 220, 221

RPCRouter, classe 220, 222
RPCRouterServlet, classe 220

S

schema, élément
  définition de types personnalisés en WSDL 191
schémas
  définition d'une table de hachage 138
  encodage de données SOAP 25–28
  XML Schema Part 2 Datatypes 32
scope (élément provider), attribut 53, 70
sécurité, SOAP et 7
send() (Message) 163
sequence, élément
  document WSDL 191
séquences d'échappement pour les caractères spéciaux 33
sérialisation 117–142
  attributs d'un élément 130
  désérialisation, valeurs multi-référencées et 29
  éléments de données 25
  encodage de types personnalisés 117–142
    classe SerialiseurMessageSourceDeDonnees 120
    collections 137–142
    tableaux épars 126–136
  racine (point de départ) pour les structures de données 43
  règles SOAP 28–31
  tableaux 38
    méthodes pour les 93
    valeurs de retour, sous GLUE 97
  types personnalisés 53
    classe BeanSerializer (Apache) 104
    conversion entre XML et Java 103
sérialisation d'objets
  désérialisation,variables multiréférencées 29
  instance de la classe Guitare (exemple) 38
  personnalisée 117–142
  règles SOAP 28–31
sérialisation personnalisée 117–142
  classe SerialiseurMessageSourceDeDonnees 120
  encodage de types personnalisés 117–142
    collections 137–142
    tableaux épars 126–136
serialiseurRequete() (SOAPMappingRegistry) 133
sérialiseurs Apache 117
SerialiseurTableauEpars, classe (exemple) 126–131
Serializable, interface 117
Serializer, interface 120
servantName, attribut 238
serveurs
  Apache Tomcat 6
    exécution avec Apache SOAP 54
  classe personnalisée OrdreBoursier_CoteServeur 100
  fautes SOAP 22
  HTTP, démarrage sous GLUE 58
  session, durée de vie d'une 74
serveurs d'applications 1
serveurs Tomcat 6
  Apache SOAP, exécution avec 54
  déploiement de service JAX-RPC sur des 241
  Java Web Services Developer Pack, utilisation avec les 233
  processeur Apache SOAP, hébergement 208
service de liste de voitures d'occasion (exemple) 174–182
  méthode de service web ajoutListe() 175
service de passerelle (exemple) 117–126
ServiceProxyCotation (exemple) 199–204
  classe RemoteCotation 201
  classe ServiceProxyCotation 202
  client GLUE pour 209
  client GLUE, développement pour un 208
  en-têtes SOAP, prise en charge 225
  interface Java IServiceProxyCotation 209
  ProxyCotation, classe (exemple) 200
service RegistreMeteo (exemple) 158
  classe RegistreMeteo 159
  client demandant des informations sur la température 166
  déploiement 160

getTemperature() 164–166
résultat RegistreMeteoResponse 167
service system/admin, GLUE 59
service, balise 238
service, éléments
document WSDL 193
ServiceBoursierDeBase, classe (exemple) 85, 86, 88, 94, 96, 111
ServiceBoursierDeBaseHelper, classe (exemple) 108
ServiceCompteurAppels (exemple) 51
descripteur de déploiement 52
ServiceDonneesEntreprise (exemple) 168–172
classe ServiceDonneesEntreprise class 169
getDonneesSymbole() 169
ServiceFauteExemple, classe (exemple) 144, 148
ServiceListeVoituresOccasion (exemple) 174–182
ajoutListe() 175
ServiceProxyAction, classe (exemple) 237
ServiceProxyActionImpl, classe (exemple) 236
ServiceProxyCotationRedirige (exemple) 226–231
classe ServiceProxyCotationRedirige 226–228
client pour le 229
déploiement 228
services d'intermédiation
.NET, pour des services basés sur GLUE 211, 213
Apache SOAP, utilisant des données d'un service basé sur GLUE 199–204
services de type messages 157–168
cotation boursière (exemple), conception d'un format de message 3
écriture 159–168
services de type RPC 11, 45–84
activation 50
clients, écriture 62–70
Apache SOAP 62–66
GLUE 66–70
déclaration sous .NET 210
déploiement 52–62
avec une portée de niveau requête 70–71
avec une portée de niveau session 72–74
sous GLUE 57–62
données d'en-tête optionnelles 50
exemple élémentaire 51
liaisons WSDL 192
noms de méthodes 46
paramètres de méthodes 47
réponses de fautes 49
réponses de méthode 48
transmission de XML utilisant un encodage littéral 168–182
attachements 173–182
types de données complexes, utilisation de 85–115
passage de tableaux en paramètres de méthodes 85–94
passage de types personnalisés en paramètres 98–109
renvoi de tableaux 94–98
renvoi de types personnalisés 109–115
URI de l'objet cible 45
services proxy
destinations finales pour 231
service de passerelle (exemple) 117–126
services web 1
.NET 210–216
approche par spécifications de type transmission 5
GLUE, informations fournies sous 61
Java Web Services Developer Pack (Java WSDP), JAX-RPC et JAXM 233
service de passerelle (exemple) 117–126
ServiceSourceDeDonnees (exemple)
classe MessageSourceDeDonnees 118
classe SerialiseurMessageSourceDeDonnees 120, 124
classe ServiceSourceDeDonnees 119
echoActions() 132
déploiement 123
servlets (Java), Apache SOAP et les 220
set() (Array) 131
setAttribute() 163
classe Element 231
setDetail() 148
setEncodingStyleURI() 171

setEncodingStyleURI() (Call) 76, 104
setHeader() (Call) 231
setHeaderEntries() (Header) 231
setMaintainSession() (SOAPHTTP-Connection) 72
setMethodName() 104
setMethodName() (Call) 63
setObject() (Value) 139
setOperationName() (Call) 217
setParams() (Call) 76
setProperty() (Call) 217
setProperty() (Context) 193
setRootPart() (repCtx) 160
setSessionTimeout() (HTTP) 74
setSOAPMappingRegistry() (Call) 104
setTargetEndpointAddress() (Call) 217
setTargetObjectURI() 104
setTargetObjectURI() (Call) 63, 193
signatures de méthodes 47
Simple Object Access Protocol (voir SOAP) xi
SOAP (Simple Object Access Protocol) xi, 5
  encodage de données 25–44
    attribut root 43
    règles de sérialisation 28–31
    schémas et espaces de noms 25–28
    terminologie 26–28
    type, indication 31–42
    valeurs par défaut 42
  espace de noms pour les enveloppes 162
  implémentations 6
    Apache 6
    Axis 7
    GLUE 7
    Microsoft .NET 7
    (voir également Apache SOAP; Axis; GLUE; .NET) 7
  liaisons, génération 204
  messages 9–23
    attribut actor 17
    attribut encodingStyle 19
    attribut mustUnderstand 18
    élément Body 20
    élément Envelope 16
    élément Header 16
    enveloppes 13–16
    fautes 21–23
    liaison HTTP 9
    réponse HTTP 11
    requêtes HTTP 10
    versions d'enveloppe 20
  sécurité, pas de traitement de la 7
  services de type messages 157–168
    écriture 159–168
  services de type RPC 45–84
    activation 50
    appel de méthode RPC, représentation dans SOAP 45–51
    déploiement avec une portée de niveau requête 70–71
    déploiement avec une portée de niveau session 72–74
    déploiement de service 52–62
    déploiement sous GLUE 57–62
    données d'en-tête optionnelles 50
    écriture de clients pour les services 62–70
    nom de la méthode invoquée 46
    paramètres de méthodes 47
    passage de paramètres 74–84
    réponses de fautes 49
    réponses de méthodes 48
    service élémentaire (exemple) 51
    URI de l'objet cible 45
  style d'encodage 157
  utilisation avec Java 5
  valeurs composées 27
soap:élément address, documents WSDL 193
  attribut location 195
soapAction, attribut 192
SOAPAction, champ d'en-tête 212
  services .NET, positionnement pour les 217
SOAPContext, classe 159, 222
SOAP-ENC qualificateur d'espace de noms 25
SOAP-ENC:attribut arrayType 38, 39
SOAP-ENC:attribut offset 41
SOAP-ENC:attribut root 43
SoapEncUtils 122
SOAP-ENV 16
  code faute côté client 146
  codes de fautes serveur 145
  élément Body 21

élément Header 16
Server fault code 143
SOAP-ENV:attribut encodingStyle 65
élément getPrix (ServicePrixActions) 77
SOAP-ENV:attribut mustUnderstand 226
SOAPException, classe 145, 152, 154
SOAPFaultEvent, classe 147
SOAPFaultListener, interface 147
SOAPHTTPConnection, classe 72
SOAPMappingRegistry, classe 104, 133, 197
SOAPTransport, classe 163
spécifications de niveau transmission Java, pour SOAP 6
spécifications de type API vs. de type transmission 4
spécifications de type transmission 5
SQL (Structured Query Language), interrogation de base de données (exemple) 41
StringWriter, classe 130
structs
réponses de méthodes RPC SOAP 48
SOAP RPC appels de méthodes 46
structures 37
stubs et ties pour les services
génération par l'outil xrpcc 237
types de données personnalisés, prise en charge de la mise en correspondance et sérialisation 242
styles d'encodage
document XML littéral avec paramètres SOAP 171
paramètres de type personnalisé 104
sérialisation de paramètres, URI pour 76
type personnalisé mis en correspondance avec une classe Java 102
suppression de
services Apache SOAP 70
services GLUE 61
syntaxe des appels de fonctions et de méthodes 2
système de messages, JAXM (API Java API pour les messages XML) xii
systèmes d'envoi de messages 2
systèmes de transfert de messages xi
systèmes de transmission de messages basés SOAP (JAXM API) 6
systèmes Windows, exécution de la commande console pour GLUE 58

T

tableaux 38–41
de chaînes dans un document WSDL 191
de cotations 207
épars 41
multidimensionnels 38
partiels 40
passage en paramètres de méthodes de services 85–94
méthodes de sérialisation 93
renvoi aux appels à des méthodes de service 94–98
tableaux d'octets, encodage 34
tableaux (de chaînes) vides 136
tableaux d'octets 34
tableaux épars 41
sérialisation 126–136
sérialiseur, ajout d'une entrée de mise en correspondance au descripteur de déploiement 132
tableaux hétérogènes
paramètre de méthode dans l'exemple ServiceBoursierDeBase 88
paramètres pour la méthode de service executeOrdre() 93
tableaux homogènes 88
tableaux multidimensionnels 38
tableaux partiels 40
tables de hachage
information sur le service Registre-Meteo 160
enregistrement de la température à une heure donnée 166
sérialisation personnalisée sous GLUE 137–142
targetAddress dans l'en-tête SOAP 226, 228, 231
targetNamespace, attribut 191, 238
élément definitions, WSDL 204
text/xml type de média, messages SOAP 10
The Mind Electric (GLUE) 7
ties, génération pour les services avec l'outil xrpcc 237

titres de nouvelles sur une société representee par un symbole d'action, service de recherche 185
toArray() (ArrayList) 218
toString() 64
  classe Liste 173, 182
  classe MinMax_CoteServeur 114
  classe personnalisée MinMax_CoteClient 111
type de contenu, spécification dans l'entête HTTP 10, 11
type variant, polymorphisme dans les 36
type, attribut 53, 161
Type, classe 137
typeNamespace, attribut 238
types de données 31–42
  accesseurs polymorphes 36
  binaires 34–36
  classe Type (GLUE) 137
  complexes, méthodes de service et valeurs de retour 85–115
    passage de types personnalisés en paramètres 98–109
    renvoi de tableaux 94–98
    renvoi de types personnalisés 109–115
    tableaux en paramètres de méthodes 85–94
  éléments primitifs, mise en correspondance avec un tableau de types Java primitifs 182
  SOAP 26
  XMLType.XSD_STRING 217
types de données complexes
  tableaux de chaînes, document de service WSDL 191
  tables de hachage 139
types de données composés 37–41
  génériques 41
  structures 37
  tableaux 38–41
    tableaux épars 41
    tableaux partiels 40
types de données personnalisés
  classe Cotation 186, 195
  dans les services WSDL 191
  éléments de données dans les documents WSDL 207
  encodage 117–142
  passage en paramètres de méthode 98–109
    correspondance de types personnalisés avec des classes Java 102
  renvoi à partir d'appels à des méthodes de service 109–115
  valeurs de retour d'une méthode de service, mise en correspondance avec la classe client Apache SOAP pour service .NET 215
  valeurs de retour d'une méthode de service, mise en correspondance vers une classe 208
types de données prédéfinis, XML Schema Part 2 32
  dans SOAP 26
types de données simples 32–34
  chaînes 33
  énumérations 33
types MIME
  algorithme d'encodage base64 pour les tableaux d'octets 34
  attachements, prise en charge par JAF 175
  messages multiparties 159

U

unmarshall() 120, 123
  classe Envelope 163
  désérialisation de tableaux épars 131
URI des objets cibles 45
URL
  client Axis, création pour un 217
  routeur SOAP pour les services de messages 163
  services GLUE 193
  vers un fichier WSDL de description de service 67
URN (universal resource name) 61
utilitaire java2wsdl 204–208

V

valeurs 26
  défaut 42
  multiréférencées 29
  types de données SOAP, simples et composés 27

valeurs 6 bits, diviser les données binaires en 35
valeurs à virgule flottante
  classe technique pour les paramètres out 81
valeurs composées 27
valeurs de paramètres par défaut
  méthodes C++ de définition 42
valeurs de retour, appels à des méthodes de service
  tableaux 94–98
  types personnalisés 109–115
valeurs hexadécimales 35
valeurs multiréférencées 29
valeurs null, vérification dans les objets lors d'une sérialisation 122
valeurs par défaut 42
valeurs simples 27
Value, classe 139
variable sous_elem 163
vecteurs
  chaînes de titres de nouvelles, stockage dans des 187
  d'éléments, detail 149
  éléments body, contenant des 160
  entrées d'en-tête, stockage dans des 231
  liste des identifiants pour le service de voitures d'occasion 177
Vector, classe 76
versions, SOAP enveloppes 20
virgules flottantes
  flottants en Java et SOAP 27

W

WAR, fichier (web application archive) 240
web application archive (fichier WAR) 240
Web Services Definition Language (voir WSDL) 61
Web Services Developer Pack (Java WSDP) 233
WEB-INF répertoire racine 240
writeElement() 139
writeObject() 137
  sérialisation du contenu d'un type personnalisé 139
Writer, classe 122
writeSchema() 137, 138
writeType() 139
WSDL (Web Services Definition Language) 7
  demande d'une description de service sous 61
  GLUE, génération automatique de la description d'un service 67
  JAX-RPC clients stubs, génération à partir de 243–245
  outil wsdl2java 184
  SOAP interopérabilité et 183–218
    appel d'un service Apache SOAP depuis un client GLUE 204–218
    appel d'un service GLUE depuis un client Apache 193–199
    service d'intermédiation utilisant Apache SOAP 199–204
    vue d'ensemble de WSDL 184–193
  utilitaire java2wsdl 204–208
  utilitaire wsdl2java 107
WSDL, documents 185

X

XML
  API GLUE pour 152
  attribut soapActionn 192
  balises, création 122
  conversion entre XML et Java 113
  descripteurs de déploiement 52–54
  données autodescriptives 4
  élément Body 13
  élément Header 16
  éléments dans les messages SOAP, indépendants ou encapsulés 28
  éléments de l'enveloppe 13
  éléments Envelope 16
  éléments service 193
  encodage de données 25
  encodage littéral 168–182
    attachements et 173–182
  enveloppe pour un service de messages 161–163
  espaces de noms 13
  fragment de schéma correspondant à une classe Java 37, 99
  JAX (Java API for XML) 233
  JAXM (Java API for XML Messaging) 246

relations entre objets 31
SOAP et xi
tableaux 38
types de données
données composées génériques 42
prédéfinis 26
Schema Part 2 32
WSDL (Web Services Definition Language) (voir WSDL) 184
XML, attributs
attribut location 195
XML, éléments
attribut actor 17
attributs (voir attributs, XML) 17
détails d'une faute 149
élément Body 13, 20
fautes 21–23
élément Envelope 13, 16
élément Fault 21
élément Header 16
qualificateurs d'espaces de noms 14
récupération depuis le corps de message SOAP 160
service 53
faultListener 53
XML, espaces de noms 13
xml2JavaClassName, attribut 103, 124
xmlns, attribut 163
xmlns, mot-clé 14
XMLParserUtils, classe 149, 162
xsd, identifiant d'espace de noms 26
xsi, identifiant d'espace de noms 26
xsi:attribut de type, affectation de ns2:Array 88

À propos de l'auteur

Robert Englander est président de MindStream Software, Inc. (*www.mindstrm.com*). Il fournit des conseils et des services quant à l'architecture logicielle, le design et le développement, tout comme il développe pour différents projets. Dans ce domaine, ses préférences vont aux architectures et aux systèmes distribués.

Rob a développé des logiciels en Java et en C++ autant pour de petites organisations que de grandes entreprises. Parmi ses nombreuses activités, il a participé à des conférences, écrit des articles et est l'auteur de *Developing Java Beans* (O'Reilly & Associates).

À propos des traducteurs

Dalil Djidel travaille chez Oracle depuis 6 ans. Il a successivement été consultant, puis responsable technique de compte. Expert technique de la suite logicielle Oracle Applications, il s'est spécialisé dans l'E-procurement et la recherche plein-texte. Lorsque le support technique lui en laisse le temps, il parfait sa culture historique et cultive une passion coupable pour les grosses motocyclettes. Il a déjà co-traduit Oracle et *XML – Développement d'applications* (O'Reilly).

Cet ouvrage a été relu par **Michel Casabianca**, développeur d'applications XML en Java.

Colophon

Notre style résulte des commentaires de nos lecteurs, de notre expérience personnelle et des réactions des divers canaux de distribution. Des couvertures spécifiques complètent notre approche particulière des sujets techniques, en insufflant personnalité et vie à des sujets potentiellement arides.

L'animal de couvertutre est une rascasse volante, *Pterois volitans*. Ce poisson vit dans les eaux chaudes et tropicales de l'Océan Indien et Pacifique à des profondeurs de 2,55 mètres. On trouve cette espèce sur les récifs, tapie dans l'ombre des grottes et des crevasses le long des tombants. La rascasse volante fait partie de la famille des *Scorpaenidae*, poisson scorpion. En effet ses nageoires latérales sont très grandes et larges se terminant par des épines flottantes. Ses voiles sont de véritables épines empoisonnées. Son corps est de couleur brune, parfois tirant sur le rouge, des barres transversales, plus claires, bordées de blanc, donnent à l'animal un aspect inquiétant. C'est un poisson court, massif et trapu de 35 cm environ en milieu naturel.

Ce poisson, calme et serein, cache un appétit féroce. Sa grande bouche lui permet de gober poissons et crustacés mal aventurés. On les rencontre parfois aux abords des récifs, en « meute », chassant leurs proies. Ils sont voraces et carnivores et n'acceptent que des proies vivantes, en particulier de jeunes poissons et des Nereis. Il vaut mieux garder un peu de distance avec ce scorpion des récifs, tout du moins ne pas le toucher.

Ce poisson peut aussi vivre en aquarium. D'une manière générale, ne descendez pas en-dessous d'un volume inférieur à 400 litres. Sa taille pourra vite avoisiner les 20 cm. Le Pterois a besoin de zones d'ombre, pas trop de courant violent. Plusieurs individus peuvent être maintenus dans un même aquarium. Par contre, les poissons plus petits et d'autres espèces seront des proies garanties. C'est un carnassier peu difficile une fois acclimaté (moules, crevettes, poissons). Par contre, il se nourrit presque exclusivement en pleine eau.

L'édition française

Emma Colby a conçu la couverture originale à partir d'une série de Edie Freedman. L'image de couverture est inspirée d'une gravure du xix^{e} siècle du « Dover Pictorial Archove ». Hanna Dyer a adapté la couverture française et Melanie Wang en a composé la maquette finale.

Les fontes du texte sont Le Monde Livre, Le Monde Sans et TheSansMono Condensed.

La mise en page de la version française a été réalisée par ***Style Informatique***.

Au catalogue O'Reilly

Java

Java en action

Ian F. Darwin
ISBN 2-84177-203-9
1re édition, avril 2002
864 pages, 58 €

> Solutions adaptables pour tous niveaux.

Java en action se présente comme une collection de solutions, de problèmes, d'exemples pratiques. Il offre aux développeurs une manne de segments de code utilisables facilement dans leurs programmes. Ce livre de recettes prend le parti de l'utilité et de l'habileté. Il dispense des segments de code qui couvrent toutes les API, même les plus spécialisées (servlets et médias) et aidera tous les développeurs déjà familiers de Java.

Servlets Java – Guide de programmeur

Jason Hunter & William Crawford
ISBN 2-84177-196-2
2e édition, décembre 2002
835 pages env., 52 €

> Guide d'apprentissage pour expérimentés et spécialistes.

Les servlets Java permettent de remplacer efficacement les scripts de CGI, et ce de manière portable, fiable et rapide. Cet ouvrage explore tout ce qu'il faut connaître pour écrire des *servlets* efficaces.

Cette seconde édition s'enrichit des nouveaux dispositifs de la version 2.2 des API des servlets Java et de nouveaux chapitres sur la sécurité et la transmission avancée. Elle détaille aussi les outils nécessaires à l'intégration de la technologie des servlets à la création de pages web dynamiques (JavaServer Pages, Tea, XMLC et Element *Construction Set*).

Java et XSLT

Eric M. Burke
ISBN 2-84177-205-5
1re édition, mars 2002
504 pages, 43 €

> Guide d'apprentissage pour expérimentés et spécialistes.

Si Java est un langage à vocation universelle, XSLT fait, lui, partie des langages spécialisés. Tout deux préparent le monde des applications réseaux.

L'alliance de ces deux langages pour le traitement de données XML offre au développeur le meilleur des deux approches. Ce livre fourmille d'exemples pour lesquels cette combinaison est idéale, sous forme d'applications « stand-alone » ou de servlets, qu'il s'agisse de mettre en œuvre des forums de discussion ou de produire des documents pour les appareils sans fil, etc.

Java in a Nutshell

David Flanagan
ISBN 2-84177-216-0
4e édition, octobre 2002
1152 pages, 54 €

> Guide d'apprentissage pour expérimentés et manuel de référence pour tous niveaux.

Cet ouvrage, écrit dans un style concis et bourré d'exemples d'utilisation, servira à la fois de guide de référence et de tutoriel pour les programmeurs avertis qui souhaitent découvrir Java rapidement.

La partie « référence » passe en revue toutes les classes des paquetages fondamentaux de Java, pour les versions 1.0, 1.1, 1.2, 1.3 et 1.4 du SDK. La partie « tutoriel » donne aux développeurs tous les éléments pour comprendre ce qui fait la spécificité d'un programme écrit en Java.

JavaServer Pages

Hans Bergsten
ISBN 2-84177-145-8
1re édition, décembre 2001
548 pages, 49 €

> Guide d'apprentissage pour tous niveaux.

Les *JavaServer Pages* (JSP) constituent un composant-clé de la plate-forme J2EE dont elles peuvent exploiter les nombreuses bibliothèques.

Avec cet ouvrage, les créateurs de sites web apprendront rapidement à utiliser les JSP dans leurs documents, afin d'interagir avec les composants serveur de l'application. Quant aux programmeurs Java, ils trouveront comment développer des composants JSP et les combiner efficacement avec d'autres technologies serveur.

Java Message Service

Richard Monson-Haefel
& David Chappell
ISBN 2-84177-208-X
1re édition, mars 2002
222 pages, 35 €

> Guide d'apprentissage et manuel de référence pour expérimentés et spécialistes.

Cet ouvrage est une introduction complète à l'API JMS. Constituante de l'environnement Enterprise JavaBeans, JMS est un service de messagerie entre applications. Il en décrit les modèles de communication, dont le modèle « point-to-point » (files d'attente) et le modèle sur abonnement.

Cet ouvrage explique en détail cette API et en permet une utilisation judicieuse.

O'REILLY®
www.oreilly.fr

Achevé d'imprimer par Corlet, Imprimeur, S.A. - 14110 Condé-sur-Noireau
N° d'Imprimeur : 62945 - Dépôt légal : janvier 2003 - *Imprimé en France*

	Débutant	Expérimenté	Spécialiste
Se former		■	■
Agir vite			
Aide-mémoire			

O'REILLY®

Java et SOAP

SOAP *(Simple Object Access Protocol)* est le dernier protocole d'une longue lignée de technologies destinées à l'informatique distribuée. SOAP, ardemment soutenu par le W3C, n'est lié à aucun langage de programmation et fait partie du monde XML. Accéder aux *services web* de type SOAP avec Java facilite l'élaboration d'applications clientes.

Cet ouvrage vise à apporter aux développeurs une compréhension générale de SOAP, dans un contexte de développement Java. De nombreux exemples étayent le propos de l'auteur qui couvre les bases de SOAP, mais aussi des aspects essentiels du développement d'applications réelles. Ainsi, parmi les sujets abordés :

- Une introduction à SOAP (les technologies liées, les implémentations Apache SOAP et GLUE).
- Les messages (l'enveloppe SOAP : les en-têtes, les éléments du corps du message, les fautes).
- L'encodage de données (les espaces de noms, la sérialisation des types de données).
- Les services de types RPC (les méthodes et les paramètres de services).
- Les types de données complexes (les tableaux, les beans).
- La sérilisation personnalisée (les outils et interfaces de programmation).
- Les fautes et les exceptions.
- L'interopérabilité et WSDL (élaborer des clients et des services à l'aide d'Apache SOAP et GLUE, l'accès aux services .NET).
- L'utilisation des en-têtes.
- JAX-RPC et JAXM (l'API Java pour RPC XML).

Cet ouvrage, écrit pour un public familier de Java, s'adresse à tous ceux qui s'intéressent à la manière d'accéder avec Java à des *services web* de type SOAP, ainsi qu'à la façon de développer, avec Java, des services SOAP.

Robert Englander est président de MindStream Software, Inc. *(www.mindstrm.com)*. Il fournit conseils et services en matière d'architecture logicielle, de design et de développement.

JAVA ET SOAP/ENGLAND
ENGLANDER
PRIX FNA
33,25 EU
PRIX EDITEU
35 EU
9 782841 772308
PROGRAMMATION
*8534 629442 01152230 044 14

ISBN 2-84177-230-6
9 782841 772308

www.oreilly.fr